conference

proceedings

2nd Workshop on Industrial Experiences with Systems Software (WIESS '02)

Boston, Massachusetts
December 8, 2002

Sponsored by
The USENIX Association
Co-sponsored by **IEEE TCOS and ACM SIGOPS**

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

2nd Workshop on Industrial Experiences with Systems Software

Boston, Massachusetts, December 2002

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
URL: http://www.usenix.org

The price is $20 for members and $30 for nonmembers.
Outside the U.S.A. and Canada, please add
$12 per copy for postage (via air printed matter).

Past USENIX WIESS Proceedings

WIESS 2000	October 2000	San Diego, CA, USA	$18/24

© 2002 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-931971-05-6

Printed in the United States of America on 50% recycled paper, 10–15% post-consumer waste.

USENIX Association

Proceedings of the 2nd Workshop on Industrial Experiences with Systems Software (WIESS '02)

December 8, 2002
Boston, Massachusetts, USA

Symposium Organizers

Program Chair

Jeff Mogul, *HP Labs Western Research Lab*

Program Committee

Edouard Bugnion, *VMware, Inc.*
Clement T. Cole, *Consultant*
Mark Stuart Day, *Cisco Systems*
Chris Demetriou, *Broadcom Corp.*
Rob Gingell, *Sun Microsystems, Inc.*
Monika Henzinger, *Google, Inc.*
John T. Kohl, *Rational Software*
Noah Mendelsohn, *IBM Corp.*
Craig Partridge, *BBN Technologies*
Doug Terry, *Microsoft Research*

Arm-twisting Committee

Gaurav Banga, *Network Appliance, Inc.*
Kevin Fall, *Intel Research*
Brett Halle, *Apple Computer, Inc.*
Andrew Hume, *AT&T Labs—Research*
Bruce Maggs, *Akamai Technologies, Inc.*
Pierre PARADINAS, *Gemplus Labs*
Rob Pike, *Bell Laboratories*
Allyn Romanow, *Cisco Systems*

Steering Committee

Valerie Issarny, *INRIA; ACM SIGOPS Vice Chair*
Michael B. Jones, *USENIX Board of Directors*
Ethan L. Miller, *University of California, Santa Cruz; IEEE-CS TCOS Chair*
Dejan S. Milojicic, *HP Labs; previous WIESS PC Chair*

The USENIX Association Staff

External Reviewers

Shel Finkelstein, *Sun Microsystems, Inc.*
Mark Manasse, *Microsoft Research*

WIESS '02: 2nd Workshop on Industrial Experiences with Systems Software

December 8, 2002
Boston, Massachusetts, USA

Message from the Program Chair

WIESS was originally conceived as "a gathering of papers and people from industry . . . featuring work 'from the trenches.'" WIESS is designed to complement conferences such as SOSP and OSDI, by providing a venue for reporting insights generated by industrial practitioners rather than researchers.

It falls to the Program Committee, and especially to the Program Chair, to set the tone of the workshop within these guidelines. My hope for WIESS '02 was to provide a forum for papers, presentations, and discussions that are primarily concerned, not with research results, but with exploiting, inspiring, or directing research. My goal, shared with the members of the PC, was to give practitioners a chance to educate researchers, and each other, about how innovative systems-software technology is deployed in actual products and services. We see WIESS as a unique opportunity for improving the connections between systems-software researchers and some of their most important customers.

Only a few members of the Program Committee are primarily engaged in research. All PC members, however, had prior experience either of publishing or of reviewing research papers, and so the PC was well qualified to evaluate both the technical merit and the industrial importance of the submitted papers.

We knew from experience with WIESS 2000 that it might be hard to attract suitable papers. Therefore I solicited help not just from members of the WIESS PC and Steering Committee, but also from an informal "arm-twisting committee," to beat the bushes for papers. However, WIESS '02 received just 13 submissions; we heard from several potential authors that they decided not to submit papers due to time constraints or concerns about protecting proprietary ideas and information. We are thus especially grateful to the authors who did submit papers, whether or not we could accept their submission.

Each paper received at least four reviews; no PC member reviewed or discussed any paper from his or her own company. We were able to discuss each submission in detail during the PC meeting. Since we did not want WIESS simply to provide yet another venue for technical papers, we rejected some papers not because of any technical flaws, but because they did not meet our unique goals for WIESS. Each of the five accepted papers was then shepherded by a PC member during the revision process.

The PC also worked together to select our keynote and invited speakers and to design the panel session. We hope that between the papers, speakers, and panelists we have provided a valuable service to the systems-software community.

I have never encountered a more thoughtful or responsible Program Committee. I would also like to thank the members of the WIESS Steering Committee and arm-twisting committee, the outside reviewers, and the sponsors of WIESS: USENIX, ACM SIGOPS, and IEEE TCOS, as well as HP Labs for sponsoring refreshments at the panel session. Finally, I could not have done this without the help of the USENIX staff and volunteers, including Peter Collinson, Barbara Freel, Jane-Ellen Long, Tara Mulligan, Jennifer Radtke, Becca Sibrack, Nick Stoughton, Catherine Vegher, Toni Veglia, Alex Walker, and Ellie Young.

Jeff Mogul
WIESS '02 Program Chair

WIESS '02
11:00 a.m.–12:30 p.m.

Session Chair:
Mark Stuart Day, *Cisco Systems*

Using End-User Latency to Manage Internet Infrastructure

J. Bradley Chen and Michael Perkowitz
Appliant, Inc.

Abstract

Performance is a requirement for all interactive applications. For Internet-based distributed applications, the need is even more acute – users have a choice about where they browse, and if a site's performance frustrates them they may never return. The goal of this paper is to demonstrate the power of end-to-end latency measurements of actual site traffic for assuring the performance of Internet applications. We briefly describe a system for collecting true end-to-end latency measurements from real site traffic and describe how such measurements can be used to improve user experience. We give examples to show how to use such measurements to evaluate site performance, pinpoint failures, and elucidate capacity issues. We argue that, as services delivered via the Internet become both more widespread and more complex, accurate measurement of site performance is of vital importance for both maintaining and improving end-user experience.

1. Introduction

Performance is critical to the success of all interactive applications. Although performance is too often neglected as an explicit requirement for distributed applications, performance problems are effectively the same as availability problems – once the user is gone, it does not matter if the system was down or simply too slow. A number of previous publications [1, 3, 5, 10, 13] argue that end-to-end [16] latency as experienced by application users is a key and often neglected measure of application performance. Unfortunately, true end-to-end latency data can be difficult to obtain, and is often no easier to analyze. This paper discusses a system that collects and analyzes data from the users of a Web-based Internet application, for the detection and diagnosis of a broad range of distributed system service problems. We have found that end-user response time information can be collected with no noticeable impact on end-user experience. Given appropriate analysis techniques, the resulting information can be used to detect and diagnose a broad range of problems across the entire content delivery chain. Such performance measurements complement usage information commonly derived from server traffic log file analysis.

Our goal in this paper is build on prior work in the area of latency analysis of interactive applications. We demonstrate the broad utility of end-user response time data in managing a Web-based service. We further describe the challenges in creating a product to track site problems and help managers detect and diagnose end-to-end service problems.

2. Related Work

Approaches to performance monitoring may be divided into two general categories: *robot-based* and *traffic-based*. Robot-based techniques measure the performance of artificial robot-generated traffic as an indicator of overall performance. Because the agent doing the measurement has control over the interaction, precise measurement of various components of the interaction may be made. Additionally, variables such as client operating system, web browser, and type of internet connection are known and controlled. Unfortunately, this advantage is also the weakness of robot-based measurement: because the client and interaction are formulaically controlled, they do not mirror the ever-changing nature of user behavior or the distribution of different user types. Differences among users can affect their experience, and robot-based measurement may miss those differences. A number of popular commercial services collect performance information based on robot data, including the Gomez Performance Network (www.gomeznetworks.com), the Perspective service from Keynote Systems (www.keynote.com), and Topaz Active Watch and Freshwater SiteSeer from Mercury Interactive (www.mercuryinteractive.com). Product offerings based on robot data include Agilent FireHunter (www.agilent.com), BMC SiteAngel (www.bmc.com), RedAlert from Keynote Systems and Freshwater SiteScope from Mercury Interactive.

Traffic-based techniques measure the performance of real user traffic in order to indicate application performance. In this case, the application must generally be instrumented to provide performance data. Traffic-based approaches accurately reflect the wide variation of end users and their differing experiences of the application. However, taking measurements from a wide variety of user types, in real time, without hurting user experience presents a significant technical

challenge. The greatest hurdle is that of successfully instrumenting the client side of the interaction in order to measure true end-to-end latency. Because of the technical hurdles, traffic-based performance management services are less common than robot-based ones. One such service is offered by WebHancer (www.webhancer.com). WebHancer's technique for measuring client-side performance is to provide an executable program that users must install. This gives WebHancer the ability to collect precise and accurate information from the user desktop. The disadvantage of this approach is that users must be persuaded to install the monitor before the site can collect performance information, possibly limiting the breadth of the system's coverage.

Though not in the performance management space, several services provide usage data – information about how users interact with a web site – by monitoring real traffic. These include WebTrends from NetIQ (www.netiq.com) and Hitbox from WebSideStory (www.websidestory.com). These services typically analyze web server logs or use lightweight JavaScript content to record basic user activity. Neither offers end-to-end latency as a metric.

3. Appliant's Approach

In designing a performance management solution, Appliant was convinced of two basic principles:

1. End-to-end latency is the most accurate indicator of user experience.
2. User demographics and behavior affect user experience.

Accordingly, we were driven to make the following demands of our solution:

1. It should measure the real experience of real users.
2. It should measure the experience of as many users as possible.

A robot-based solution was ruled out by the necessity of correlating user demographics with user experience. Similarly, any solution that required users to explicitly install software – an activity many were likely to find irksome or even threatening – was ruled out.

Our approach instead was to annotate web pages with JavaScript code that would perform the necessary measurements. Because the script is loaded with the page, it executes on the client and can record when a page is first downloaded, when images are loaded, and when the page completes. Page annotation requires only that the site modify their pages (typically, modifying only site-wide templates), with no installation required of users. Because JavaScript is a widely adopted internet standard, most browsers support it, ensuring data from most of a site's visitors. In this section we discuss some of the technical hurdles facing this approach and present our methodology in more detail.

3.1. Data Collection

Appliant's browser monitor uses annotations to HTML documents that enable the Web content to monitor itself as it is rendered. The annotation consists of an HTML SCRIPT tag with an include statement that loads Appliant's JavaScript subroutines. The use of an include file simplifies the Web page annotation by shortening the annotation. It also makes it possible for the JavaScript to be cached by the browser. Caching helps minimize Internet Service Provider (ISP) and Content Delivery Network (CDN) fees, and minimizes the overhead for loading JavaScript. The annotation and JavaScript code is part of the page requested by the end user, and is unloaded along with the rest of the page when the browser loads a new document.

The JavaScript code consists of state management routines plus a collection of event handlers that capture various events provided by the Document Object Model (DOM) [6, 18] to recognize and record state transitions in the browser. The Appliant routines install event handlers for the following JavaScript events:

- *onAbort* – loading interrupted
- *onError* – an error occurred while loading an image
- *onLoad* – a document or image has finished loading
- *onReadyStateChange* – the ready state of a document or image has changed
- *onStop* – loading of a web page is stopped by user

Care must be taken to insure that Appliant event handlers pass events through to any handlers previously registered by the page. Yet another problem is handlers that fail to pass events through to the Appliant handlers. This problem is common enough that we support a monitor option that uses a periodic JavaScript timer interrupt to confirm that the Appliant event handlers are registered within the browser.

The browser monitor measures *fetch time* as the time to load the HTML document, and *render time* as the time to fully render that document within the browser

environment. We report render time as beginning when the last byte of the HTML document is delivered to the browser, and ending when the *onload* event fires indicating the page has fully loaded, including the time necessary to load and render any included images. Although this simplification ignores parallelism in the browser it simplifies interpretation of the data and effectively reveals cases where render rather than fetch is responsible for poor response time.

Conceptually, fetch time begins when a hyperlink is selected to be loaded in the browser. In practice, it is not always possible to begin measurement at that instant. In a worst-case scenario, since our browser monitor is not resident in the web browser, the first chance it has to record a time stamp for the beginning of the fetch time is when the top of the HTML document arrives in the browser. In our preferred deployment technique, a cookie is used to retain a timestamp from the unload event of the previous page, which occurs when the user clicks on a link. In this way our monitor can provide very accurate fetch time measurements for consecutive requests from the same managed web site.

Even without using cookies, the fetch time measurement can be very effective. Dynamic web sites typically send a block of static HTML first, allowing the browser to get started while other dynamic components are being computed. If the entire page arrives quickly, the fetch time measurement will be inaccurate in a relative sense, although in an absolute sense the page was very fast and performance is a non-issue. If the page is delayed by slow content generation, the fetch time measurement captures that slowness, and the relative error is small. If the page is slowed due to network delays, our fetch time measurement will reflect that slowness unless the slowness is isolated to connection setup. Overall, we have found that our slight handicap in measuring connection setup time is more than offset by the benefits of complete data, and rarely interferes with our ability to identify problems with end-to-end performance.

There are many opportunities for enhancing browsers to be better sources of performance management data. Currently we are able to capture response time for each image request, but the browser does not expose whether the image was loaded from the browser cache or retrieved via the network. Another handicap is that the precise size of an HTML document is difficult to determine reliably. Such additional data from the browser could substantially improve the quality of performance data we collect.

Once monitoring of a page is complete, a data record is transferred to the data manager using a standard HTTP request. Complete data records are received by the data manager, which generally resides in the same facility as the servers for the managed web site. The data is processed either daily or in real time, depending on the Appliant product. The analysis subsystem generates summaries that support multiple reporting systems. This enables customers to access the information in various forms, including canned reports via email, an interactive Excel workbook, and a Web front-end to a database.

Browser-side filtering of data is available, based on response time thresholds, connection type or other properties of the performance event. The browser monitor also supports statistical sampling. This permits our customers to manage the capacity of their data management system, and greatly simplifies deployment on high-traffic sites.

3.2. Data Analysis

Given the potentially large volume of performance data resulting from end-user monitoring, the next challenge is data collection. Realities such as firewalls oblige us to stick with the well-established standards for Web data transport to the desktop: HTTP/TCP/IP over port 80. Appliant systems use a centralized data collection scheme layered over a simple Web server. Though there are no theoretical problems to solve, there are many practical challenges in delivering a product that reliably handles hundreds or thousands of megabytes of response time data on a daily basis. More specifically, familiar challenges arise in terms of making the system scalable and reliably available. Similar challenges related to high data volume occur in data analysis, but nothing unique to our system. Web log collation and analysis provides a relevant example of strategies for processing similar volumes of data. We apply standard approaches such as cluster-based parallelism and statistical sampling to address these system-level challenges.

An interesting problem that arises in the analysis process is treatment of 'noise' and outliers in the data stream. A small amount of erroneous data is not impossible in distributed systems that handle millions of records per day over WAN TCP/IP connections [17]. Data inaccuracies can also occur due to phenomena such as defects in browser implementations, local clock adjustments that occur during a measurement, and connections that operate at effective speeds of tens of bytes per second. A more fundamental problem is that wide-area network response time data commonly

exhibits an asymmetric heavy-tailed distribution [14], for which the mean is not a robust statistic. For such non-normal distributions, a trimmed-mean is a common alternative measure that provides a more robust measure of central tendency than the mean. [8, 12] A trimmed mean is computed by excluding the extreme n% of data points, where a common value for n is 5%. Although this would provide a robust mean, identifying the 5th percentile is computationally equivalent to a median computation, which requires *O(n)* space, too expensive for the large data sets handled by our system. Instead we simply exclude values that exceed a predefined "trim" threshold, and then confirm that the amount of data trimmed was less than 5%. By default the system trims data points with response time exceeding 180 seconds. To date, simple strategies such as these have been adequate to cope with the noise inherent in end-user data.

Clock skew between the web server, browser client, and data collector is a potential problem that was avoided through the Appliant system design. A reality of the Internet and the diversity of servers it hosts is that clock skew is a common phenomenon – we routinely encounter clocks skewed by hours or days even on high-traffic, professionally managed sites. In recognition of this reality we were deliberate in creating a design that tolerates skewed clocks. The basic strategy is that timestamps are assigned only on a small number of dedicated servers within the host data-center. Desktop machines do generate latency measurements, but they do not assign time stamps. Clock skew is not an issue for latency measurements since both the start and stop time are measured relative to the same clock.

4. Analysis Examples

4.1. ISP and CDN Performance Profiling

Performance is central to the value proposition for ISP and CDN services. Without good performance data, it can be difficult for CDNs to justify the value of their services and for customers to know that they are getting what they paid for. The collection and analysis of end-user latency data makes it possible to obtain ISP and CDN performance information of exceptional detail. Figure 1 gives an example for an Internet community site, showing performance improvements during the first two weeks of CDN service for an example Web site. In this case, the potential CDN customer was skeptical about the CDN's ability to improve performance for dialup customers. The CDN used this information to demonstrate that performance improvements of 32-41% were achieved for the customer's site across all classes of end-users.

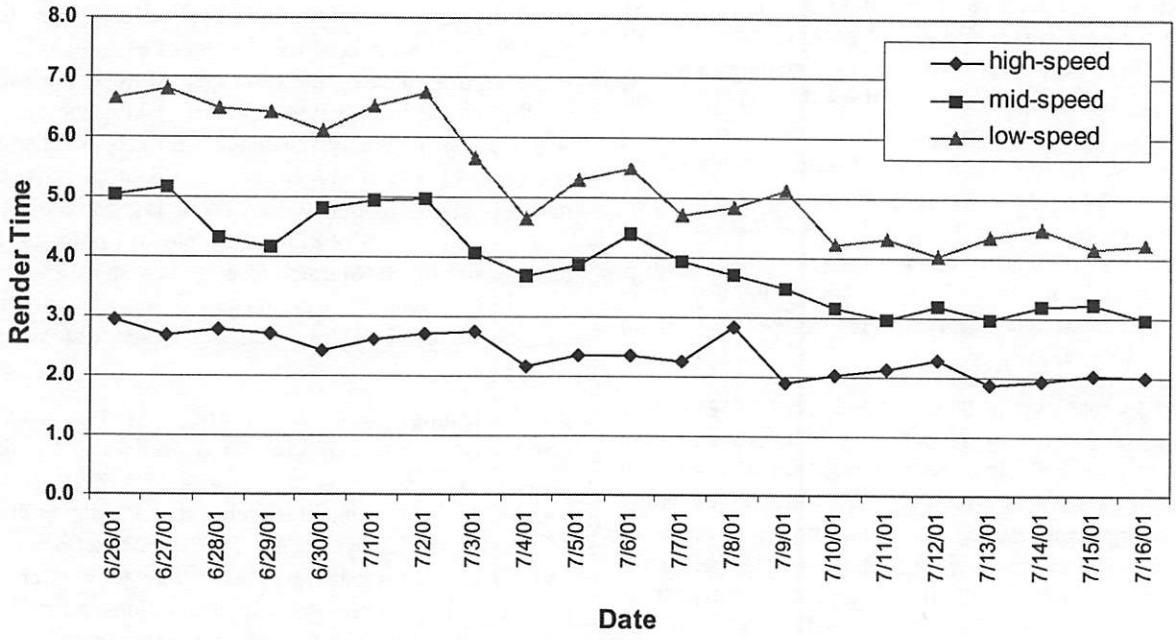


Figure 1. Documenting CDN Performance Improvements. CDN service began on 28 June 2001.

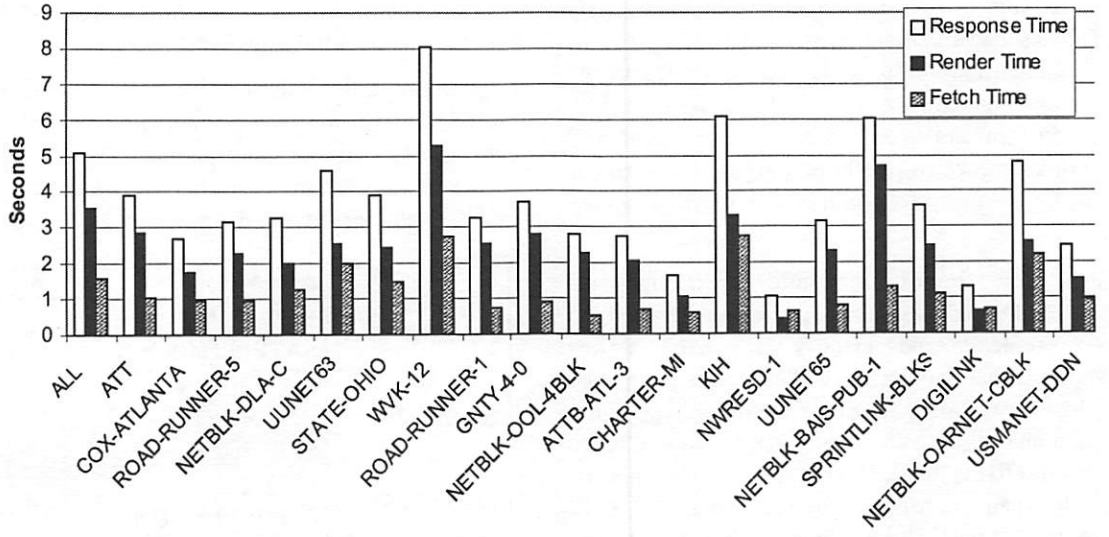


Figure 2. Response Time for Busiest ARIN Netblocks, US LAN traffic only.

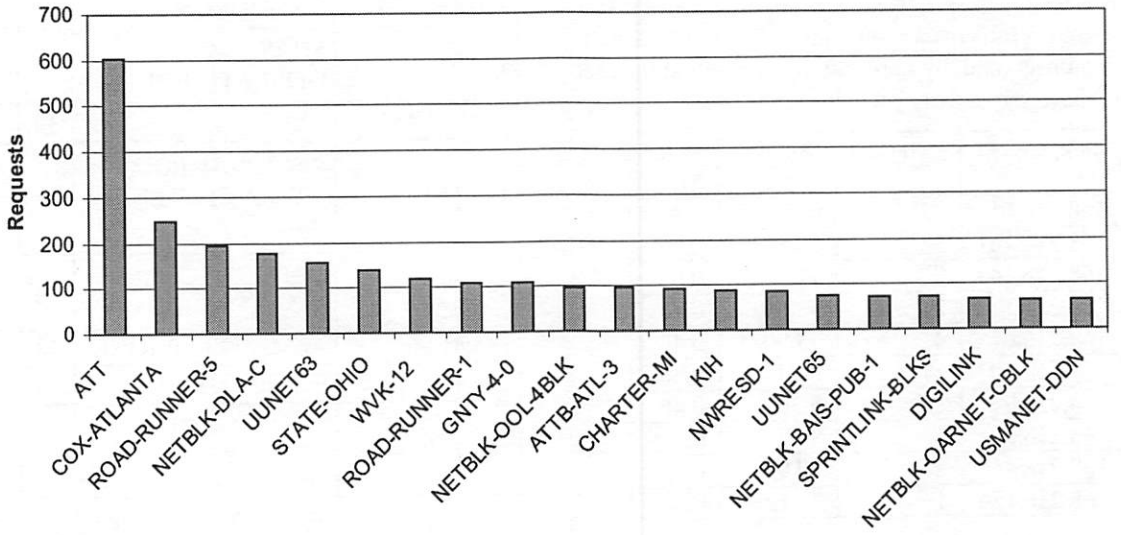


Figure 3. Traffic for Busiest ARIN Netblocks. US LAN traffic only.

Even when ISPs and CDNs deliver good performance, they do not provide equal benefit to all Internet users. Performance data from end-users can be used to quantify site performance all the way to the desktop, thereby identifying network subnets that get the best and the worst service. Figure 2 shows response time in terms of fetch time and render time for an Internet sports site, for the busiest IP address ranges as defined by ARIN *whois* data[1]. Figure 2 shows US LAN-traffic only[2], as desktop connection type has a significant impact on end-to-end response time. The figure shows below-average performance for netblocks maintained by the West Virginia K-12 system (WVK-12), Kentucky Department of Information Systems, and Bell Atlantic (now administered by Verizon). Together with Figure 3, showing traffic levels for the same address

[1] See ftp.arin.net and www.arin.net for more information.

[2] Connection type is obtained using the "clientCaps" facility in Microsoft Internet Explorer. For more information see http://msdn.microsoft.com/workshop/author/behaviors/reference/behaviors/clientcaps.asp.

ranges, this information enables a web site to identify the Internet subnets that are most important in terms of traffic volume, and compare their performance to the site average (given by the ALL category). This information can indicate situations where poor ISP peering relationships or poorly performing CDN caches are causing performance problems for specific end-users.

Although these figures are useful for a high-level analysis, additional detail is generally required to validate problems and identify a cause. A key observation is that the blocks of IP addresses defined by whois data vary in size, and some are quite large. For a finer-grain analysis, we commonly use a mapping based on 24-bit prefixes. Table 1 shows WVK-12 subnets, ordered by their contribution to the overall WVK-12 mean response time. These subnets include the slowest WVK-12 subnet (168.216.124), and the fastest (168.216.107). Assuming that the WVK-12 subnets share the same path(s) to the Internet, the variance in performance across these subnets implies that performance problems are internal to WVK-12, as their fastest subnets (and, by implication, all external Internet factors) have satisfactory behavior.

Independent	Requests	Subnet RT	delta Mean
168.216.51	33	6.45	1.84
168.216.161	3	28.83	0.75
168.216.141	7	11.48	0.69
168.216.123	2	33.65	0.58
168.216.67	4	13.90	0.48
168.216.107	17	3.14	0.46
168.216.94	10	5.30	0.46
168.216.56	6	8.07	0.42
168.216.48	2	20.90	0.36
168.216.124	1	38.12	0.33

Table 1. WVK-12 Subnets with largest contribution to Mean.

As a final example of finer-grained data, Table 2 shows response time for the slowest 24-bit subnets with at least 0.1% of total site traffic. We note that "whois" data can often identify these subnets with surprising precision. In this table, 216.236.222 is a satellite services provider, and 134.134.248 is an Intel facility in Santa Clara, CA. Making effective use of this level of detail remains a challenge. Specific challenges include:

- Achieving a classification of appropriate granularity
- Isolating ISPs such as AOL and NewSkies that skew results due to atypical network properties
- Distinguishing "last-mile" problems from "middle-mile" problems

Regrettably, due to these challenges, solving network problems remains in the domain of "networking experts." The goal of our continuing work is to further automate the process of recognizing performance incidents and identifying them to their authentic source.

Subnet		*n*	RT
216.236.222	NEW SKIES SATELLITES N.V.	15	47.0
207.108.252	U S WEST INTERNET SERVICES	17	23.5
134.134.248	INTEL CORPORATION	20	21.2
206.129.0	N2H2	12	14.3
209.133.187	STATE OF SOUTH CAROLINA	14	10.2
170.158.130	ONONDAGA BOCES	16	9.7
12.96.122	SHASTA COUNTY OFFICE OF EDU.	21	9.7
204.171.48	VERIO	14	9.6
198.110.59	REGIONAL EDU. MEDIA CENTER 4 (MI)	16	9.3
24.187.188	OPTIMUM ONLINE (CABLEVISION)	14	9.1
216.81.96	ALMA TELEPHONE	12	8.6
209.124.103	AMNET US	23	8.5
205.154.229	GROSSMONT UNION H.S. DISTRICT	23	7.7
207.70.63	MICRON INTERNET SERVICES	18	7.5
206.78.5	VISALIA UNIFIED	14	7.1
150.176.63	FLORIDA INFO. RESOURCES NET	26	6.8
65.204.164	BBG COMMUNICATIONS	32	6.6
209.205.203	PACNET	26	6.6
216.229.196	MISSISSIPPI DEPT. OF EDUCATION	19	6.5
168.216.51	WV DEPARTMENT OF EDUCATION	33	6.5

Table 2. Performance Detail by Class-C subnet. US LAN traffic only, for subnets with at least 0.1% site traffic.

4.2. Capacity Planning

To effectively plan site capacity, a site administrator must be able to answer a number of questions:

- What is the capacity of the system?
- What is the offered load on the system?
- What resource or resources are the bottlenecks?

Examples of potential bottlenecks include CPU cycles, physical memory, memory bandwidth, IO bandwidth, and LAN bandwidth. Bottlenecks can occur anywhere in the system: Web servers, application servers, database servers, or the network. When peak demand exceeds system capacity, the result is performance problems, system failures, or both.

End-user performance data is a very effective way to answer these questions. Figure 4 shows a chart of page requests by hour over a day for an example web site. It shows a common pattern of traffic, with a peak in traffic in the daytime and a corresponding trough at night. The response time chart (Figure 5) and page request charts have different shapes, indicating that

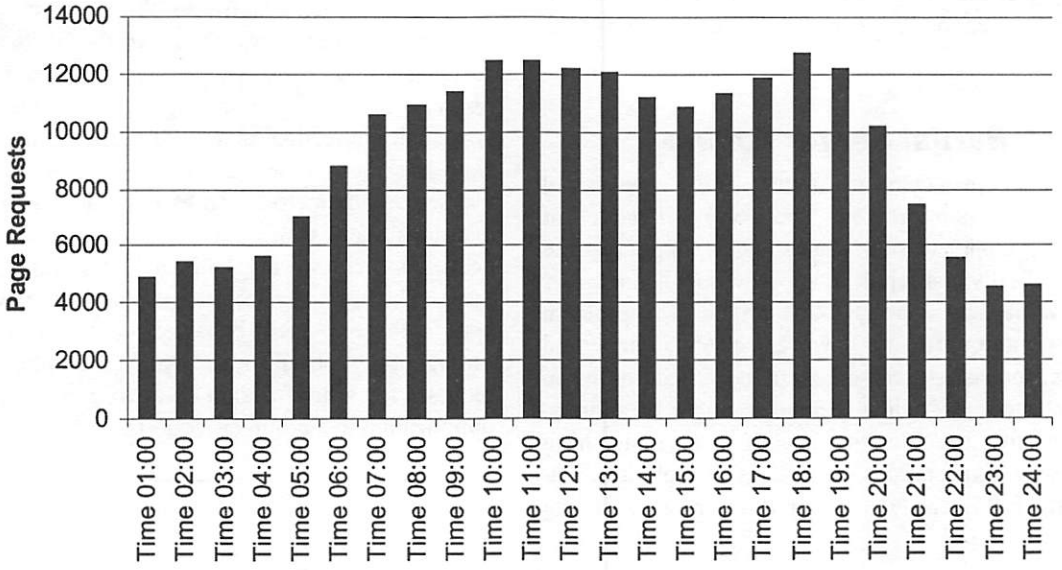


Figure 4. Page Requests vs. Time over 24 Hours

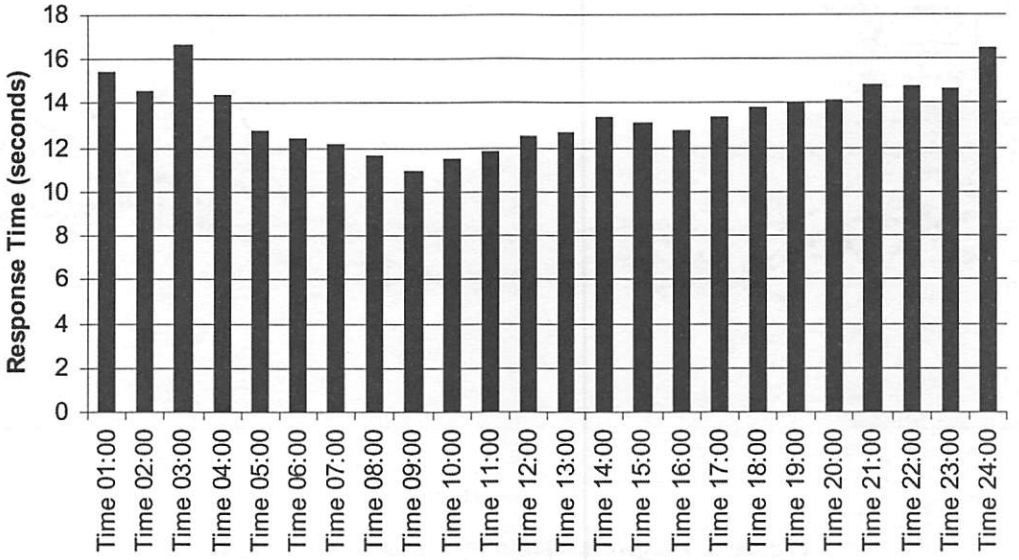


Figure 5. Site Load vs. Performance over 24 Hours

there is not a direct relationship between response time and load. In this case the response time curve shows a common pattern, with a faster average response time when many users access the site from their fast work connections, and slower traffic when more people use dialups at home. If response time tracked page requests, it would indicate a direct relationship between performance and load, evidence that an internal performance bottleneck was being stressed. The independence of response time and load in these charts indicates that the site has excess capacity, even at peak hours. If it were the case that response time tracked page requests, we could identify the bottleneck by examining additional resource charts to discover where resource utilization is correlated with performance problems.

4.3. A Partial Server Failure

Availability management tools and robot-based performance monitors can be very effective at identifying complete site failures. They simply test the site periodically and report as appropriate when a server or the whole site appears to be offline. The problem becomes more difficult, however, when a large and complex content system needs to be tested, or when server clusters are hidden behind a single IP address. When a site becomes sufficiently mature to have performance expectations as well as availability, end-user data can identify a much broader class of site problems.

As an example, Figure 6 illustrates response time by server for a cluster of servers over a 24-hour period for a web-based news site. At 6:00 AM, for example, most servers are delivering similar performance of about 6 seconds average response time, but one server (server 5) has response time of about two seconds slower. Across the 24-hour day, the same server was consistently about two seconds slower. Such problems can be extremely difficult to detect with robot-based data sources.

Considering request rates for the same time period, Figure 7 shows two kinds of servers. The bottom server (server 1) is a four-way multiprocessor. It receives about half as much traffic as the eight-way machines along the higher curve. The surprise is that the top server is the same eight-way system (server 5) that had response time problems in Figure 6. The load balancer is actually directing about 20% more traffic to the slowest server in the cluster. Load balancers generally use server-side metric such as CPU utilization to make decisions about distribution of load. In this case those decisions were wrong. Ultimately the explanation for these behaviors was that the deviant server was using beta software. Such behavior could be caused by throughput optimizations, for example, delaying requests for a brief period so-as to apply a scheduling optimization to a group or requests.

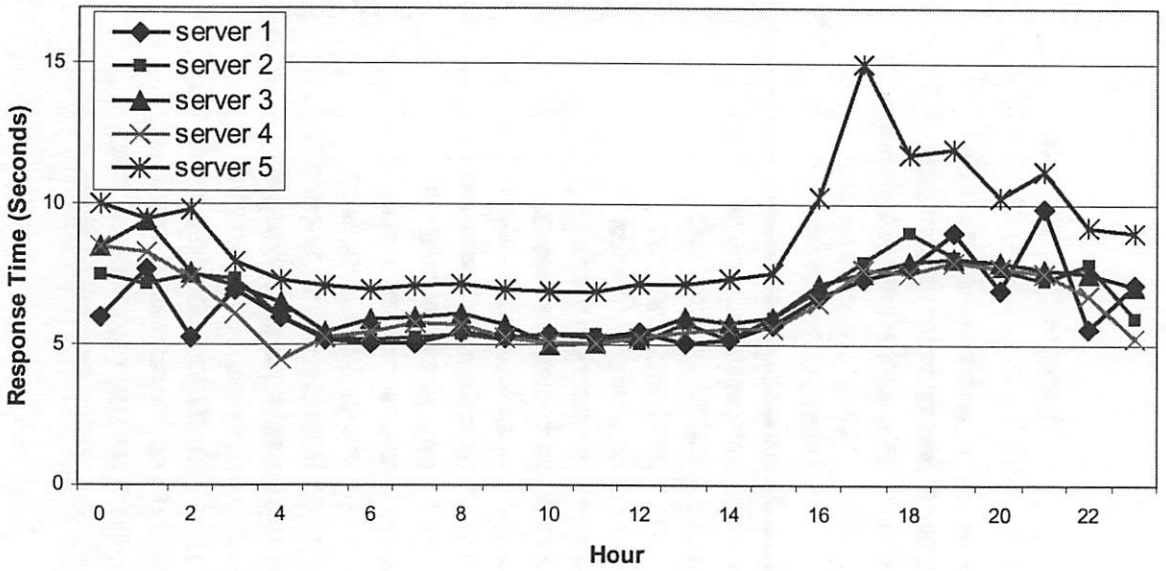


Figure 6. Response Time by Server over 24 Hours

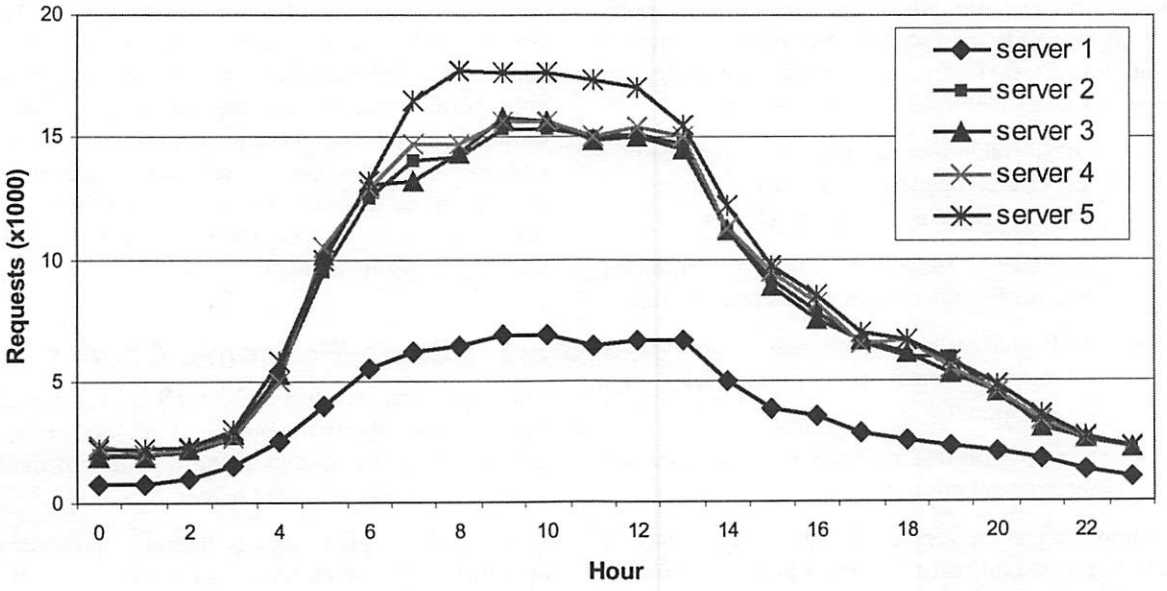


Figure 7. Page Requests by Server over 24 hours

5. Discussion

The direct presentation of our technical results obscures many of the challenges behind the development of the software we describe in this paper. Indeed this system had its share of challenges. In this section we document some of the technically relevant ones.

Although the results in this paper relied exclusively on data collected via JavaScript content annotations, considerable effort went into alternative approaches to data collection that we ultimately determined were not suitable for our company or products. We briefly considered building a network packet sniffer, such as that used by Wolman et al. [19]. Although this approach has many nice properties in terms of completeness and accuracy, a number of practical considerations caused us to exclude it early on, in particular:

- Business requirement of a software (not hardware) product
- Lack of in-house expertise
- Inability of sniffers to manage 3rd party content and desktop errors
- Anticipated performance and scalability issues

These factors led us to exclude a packet monitor early in our design process.

In addition to the JavaScript-based browser monitor, early Appliant web-management systems also used a server-side software component to measure application-level response time plus system-level performance metrics (such as CPU and memory statistics) from the server perspective. The anticipated benefits included a more precise measurement of server-side latency for content generation, plus additional information for diagnosing problems within a server. Initially we found this component a challenge to support, although the support burden became easily manageable after the bumps of early releases. Ultimately we decided to remove this monitor as a feature for the following reasons:

- Considerable cost for developing and testing a software component on a large matrix of OS/Web Server platforms
- Customer resistance against installing an executable software component on their production systems
- Marginal data value, as response time is available from desktop, and system-level metrics are provided by management frameworks.

As we describe in Section 4.3, powerful server diagnosis is possible without a server-side software install.

Another option we considered and then dismissed is a client-side installation of executable code. The potential benefits of such a configuration are significant. The stability and richness of the OS APIs could provide more precise measurements, a richer set of metrics, and more complete measurements as well. Many of these

benefits derive from the fact that an executable component avoids the security and privacy restrictions of the browser environment. Ultimately we made little use of this approach for the following reasons:

- Considerable cost for developing and testing a software component on an exceptionally large matrix of OS/Web browser platforms
- Customer resistance against installing executable software components on desktops
- Additional metrics are of marginal incremental value over data already available from JavaScript monitor
- Use of a desktop executable raises many new security and privacy issues

As compelling as the results we have achieved may be, it can fairly be said that the hardest problems have not been solved. Some of the key challenges we are considering in our ongoing work include:

- Reliably recognize abnormal behavior
- Reliably ignoring non-problems
- Classifying problems by impact and significance
- Combine multiple performance measurements into a single indication of the root cause

5.1. Combining Performance and Other Data

Application performance has a direct impact on user satisfaction, but performance data alone is not sufficient to show this relationship. By correlating performance data with other information illustrative of customer satisfaction or bottom-line success, we can determine how tolerant users are and how important performance improvements will be. Useful additional data can include user behavior or sales figures. For example, we may wish to find the correlation between average performance and user session length; do users who experience poor performance spend less time on the site? Another option would be to examine user behavior immediately following a slow page-load; do users give up and depart when a page takes a long time to load? A third possibility would be to look at the correlation between performance and conversion rate – the probability that a user will make a purchase.

Such analysis is, of course, useful in justifying the importance of good application performance, but it can also be used to prioritize infrastructure improvements. Users tend to be more tolerant of some problems than others, and limited infrastructure dollars can be channeled to the most irksome problems. For example, poor performance on catalog pages may discourage browsers from ever becoming buyers, but once a customer has committed to the checkout process, he is less likely to abandon his session because of poor performance. In this case, catalog infrastructure should receive priority attention.

5.2. Social/Economic Challenges

There are many practical problems that have substantial impact on our ability to deploy our solution and on the feature set we are able to support. A few of the most common problems are noted here.

Our solutions give strong Internet businesses an opportunity to tune and focus the effectiveness of their site. Just as high-end retailers use customer experience to differentiate themselves from discounters, we believe high-end web sites will use customer experience to improve the experience for their users, and thereby improve their bottom line. Unfortunately, many Web sites are not ready for this level of service delivery. They are overwhelmed with the struggle to keep systems running, manage content updates, while coping with tight and shrinking budgets. These problems are exacerbated by software churn, which tends to prevent systems from reaching maturity. As a result, many online services are unwilling to monitor or assure end-to-end service levels.

Another social challenge relates to privacy issues. At a technical level we believe our position on privacy is very strong. We assume that the managed Web system generates or is able to generate log files for site usage analysis. Given this assumption Appliant systems do not change the balance of privacy for end users, because they collect no additional information on behavior. They do augment existing data with performance and error information for diagnostic purposes. All data is co-located with the managed web site and under the control of the site administrator. Since the site administrator already has access to complete usage data for their site, the privacy balance remains unchanged. The product does not require cookies, although it can make use of cookies that are already used by the site. The product does not support or require the collection of zip codes, address information or account numbers of any kind.

Although our position is quite defensible at a technical level, prospective customers are frequently unable or unwilling to consider the technical details. Frequently

they are obliged to take a very conservative position on privacy: that any additional data collection is unacceptable, regardless of the fact that additional data is of diagnostic value only and does not reveal information about individual users. For these sites, an opt-out program can be used to achieve an acceptable solution.

5.3. Prior Research in Latency

In the research community, the argument for use of latency as a measure of system performance emerged from prior concerns about the effectiveness of micro-benchmarks and throughput benchmarks for measuring system performance [1, 10, 11, 13]. Endo et al. made a direct argument advocating such techniques [5]. Whereas the Endo study was limited to interactive applications on a standalone desktop, our interest is in interactive distributed applications on the Internet. Since that publication, relatively little work has been done on the subject, with the bulk of the work occurring in the context of real-time but non-interactive applications such as streaming media. Flautner et al. [7] included latency-based analyses to show that real-time apps benefit less from threading on a multiprocessor than on a uniprocessor (15% vs. 4%). Jones and Regehr [9] used latency measurements to analyze thread-scheduling issues in support of real-time applications.

We note two prior publications that study latency in the context of the Web. Wolman et al. [19] studied Web end-user latency in the context of cooperative Web proxy cache performance. Ramakrishnan and Elnozahy [15] describe a system for collecting response time on end-user browsers that has some similarities to ours, particularly in the raw data that is collected. Although this paper clearly predates the present work, we believe that our systems were developed and released as products prior to theirs. The Ramakrishnan paper includes an ample discussion of the data collection methodology on the client, along with one example of analysis for a small Internet Web site. The present paper builds on the Ramakrishnan publication by documenting challenges in working with large production Web sites, in contrast to the relatively small site used by Ramakrishnan. Further, our experience in working with response time data allows us to explore in more detail the exceptional diagnostic power of end-user information.

6. Conclusions

The performance of web sites and Internet applications varies widely and is often the main factor in determining the quality of the end-user experience. End-to-end response time is a key measure of performance, and we believe it to be a vital differentiator for end users. Response time measurements, moreover, are an important potential resource for site development and operations. In this paper we have demonstrated how such information, derived from real traffic, can be used to systematically analyze situations that are very difficult to detect or diagnose without such measurements.

The systems described in this paper anticipate a day when Web browsers and other end-points for Internet applications expose detailed, accurate service quality metrics through stable APIs. Ideally such measurements would include end-to-end application-level performance measurements, error detection, and failure reporting. Although Appliant systems can provide this information for the current Web infrastructure, the lack of stable, consistent management APIs complicates the implementation. We believe that the importance of service level monitoring systems will increase as Web services mature. In this context, outsourced infrastructure and integration with third-party services becomes more common, and their performance becomes a requirement and a success factor. This is in contrast to the status quo, where distributed application performance is often considered only after deployment.

Acknowledgements

We would like to acknowledge the hard work and dedication of the Appliant product and development team. The results described in this paper are a direct result of their commitment to fidelity in distributed application performance management. We would also like to thank Craig Partridge and the WIESS Program Committee for their comments on preliminary drafts of this paper.

Author Contact Information

Please note preferred email addresses for the authors as follows:

J. Bradley Chen: brad.chen@acm.org

Mike Perkowitz: mike@perkowitz.net

7. References

1. Brian N. Bershad, Richard P. Draves, and Alessandro Forin, "Using Microbenchmarks to Evaluate System Performance." *Proceedings of the Third Workshop on Workstation Operating Systems*, IEEE Computer Society, Los Alamitos CA, April 1992.
2. J. Bradley Chen, "SLA Promises," Appliant Technical Note, June 2002. Available from http://www.appliant.com/techresource/techresource.php.
3. Erik Cota-Robles and James P. Held, "A Performance of Windows Driver Model Latency Performance on Windows NT and Windows 98," *Third Symposium on Operating System Design and Implementation*, USENIX Association, Berkeley CA, February 1999.
4. Kenneth Duda and David Cheriton, "Borrowed-Virtual-Time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose schedule." *Seventeenth ACM Symposium on Operating System Principles*, ACM, New York NY, December 1999.
5. Yasuhiro Endo, Zheng Wang, J. Bradley Chen, and Margo Seltzer, "Using Latency to Evaluate Interactive System Performance." *Second Symposium on Operating System Design and Implementation*, USENIX Association, Berkeley, CA, October 1996.
6. David Flanagan, "JavaScript, The Definitive Guide, Third Edition." O'Reilly and Associates, Sebastopol, CA, 1998.
7. Kriztian Flautner, Rich Uhlig, Steve Reinhardt, and Trevor Mudge, "Thread-Level Parallelism and Interactive Performance of Desktop Applications," *Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York NY, November 2000.
8. David C. Hoaglin, Frederick Mosteller, John W. Tukey, *Understanding Robust and Exploratory Data Analysis*. John Wiley and Sons, Hoboken NJ, 2000.
9. Mike Jones and John Regehr, "The Problems You're Having May Not Be the Problems You Think You're Having." *The Seventh Symposium on Hot Topics in Operating Systems*, IEEE Computer Society, Los Alamitos, CA, March 1999.
10. Jeffrey C. Mogul, "SPECmarks are leading us astray." *Proceedings of the Third Workshop on Workstation Operating Systems*, IEEE Computer Society, Los Alamitos CA, April 1992.
11. Jeffrey C. Mogul, "Brittle metrics in operating systems research." *The Seventh Symposium on Hot Topics in Operating Systems*, IEEE Computer Society, Los Alamitos, CA, March 1999.
12. NIST/SEMATECH e-Handbook of Statistical Methods, http://www.itl.nist.gov/div898/handbook/, September 2002.
13. John Ousterhaut, "Why Operating Systems Aren't Getting Faster As Fast As Hardware." *Proceedings of the Summer 1991 USENIX Conference*, USENIX Association, Berkeley CA, June 1991.
14. Vern Paxson and Sally Floyd, "Wide Area Traffic: The Failure of Poisson Modeling." *IEEE/ACM Transactions on Networking*, Volume 3, Number 3, 1995, pg 226-244.
15. Ramakrishnan Rajamony and Mootaz Elnozahy, "Measuring Client-Perceived Response Times on the WWW." *USENIX Symposium on Internet Technology and Systems*, USENIX Association, Berkeley, CA, March 2001.
16. Jerome H. Saltzer, David P. Reed, and David D. Clark, "End-to-End Arguments in System Design," *ACM Transactions in Computer Systems*, Volume 2, Number 4. ACM, New York NY, November 1984
17. Jonathan Stone, Michael Greenwald, Craig Partridge, and James Hughes, "Performance of Checksums and CRCs over Real Data." *IEEE/ACM Transactions on Networking*, Volume 6, Number 5, ACM, New York NY, Oct 1998.
18. World Wide Web Consortium, "Document Object Model (DOM) Level 2 HTML Specification, Version 1.0." Candidate Recommendation, June 2002. Available from http://www.w3.org/TR/2002/CR-DOM-Level-2-HTML-20020605
19. Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy, "On the Scale and Performance of Cooperative Web Proxy Caching." *Seventeenth ACM Symposium on Operating System Principles*, ACM, New York NY, December 1999.

Building an "impossible" verifier on a Java Card*

Damien Deville
deville@lifl.fr, http://www.lifl.fr/~deville
Gilles Grimaud
grimaud@lifl.fr, http://www.lifl.fr/~grimaud
Université des Sciences et Technologies de Lille,
Laboratoire Lifl, Bat M3,
cité scientifique, 59655 Villeneuve d'Ascq - France

Abstract

Java is a popular development platform for mobile code systems. It ensures application portability and mobility for a variety of systems, while providing strong security features. The intermediate code (byte code) allows the virtual machine to verify statically (during the loading phase) that the program is well-behaved. This is done by a software security module called the *byte code verifier*. Smart Cards that provide a Java Virtual Machine, called Java Card, are not supplied with such a verifier because of its complexity. Alternatives are being studied to provide the same functionality outside the card. In the present paper, we propose to integrate the whole verifier inside the smart card. This ensures that the smart card becomes entirely autonomous, which allows full realization of smart cards potential as pervasive computing devices. Our verifier uses a specialized encoding and a software cache with a variety of cache polices to adapt to the hardware constraints of smart card. Our experimental results confirm the feasibility of such a security system being implemented in a smart card.

Keywords: Smart card, Java Card, static verification, type inference, secure embedded system.

*This work was supported by a grant from Gemplus Research Labs, the CPER Nord-Pas-de-Calais (France) TACT LOMC C21, under the European IST Project MATISSE number IST-1999-11435, and our prototype was performed within the Gemplus Research Labs and considred by our partner as a proof of concept for further industrial development.

1 Introduction

This paper presents the motivations and techniques used to develop a stand-alone verifier inside a smart card. The verification process is one of the most important parts of the security in a Java Virtual Machine. In a JVM, this process is performed while a new application is loaded into the virtual machine. Due to smart card constraints, a verifier as it was defined by Sun is said to be impossible to embed in a Java Card Virtual Machine (JCVM).

"Bytecode verification as it is done for Web Applet is a complex and expensive process, requiring large amount of working memory, and therefore believed to be impossible to implement on a smart card" [12].

"Clearly, bytecode verification is not realizable on a small system in its current form. It is commonly assumed that downloading bytecode to a JAVACARD *requires that verification is performed off-card"* [18].

Nevertheless, we will present here some solutions to obtain a stand-alone verifier in a JCVM.

We first outline the standard verification process, in order to introduce the reader to standard verification principles. Next we present degraded solutions of this process that have been designed to cope with supposed smart card constraints. To overcome these difficulties, we show that these constraints are not actually a problem by detailing the properties of smart card hardware properties used for our stand-alone verifier. We then present our approach that consists of hadrware-

specific adaptations to efficiently match the standard algorithm with these detailed hardware characteristics. We do not change the standard verification algorithm, but instead propose some innovative techniques that allow an improved execution on top of the smart card limited hardware. In the last part we give some experimental results extracted from our prototype. Our prototype subsequently became a proof of concept of an embedded smart card verifier for our industrial partner.

2 Verification principles

First we present the standard verification process performed by the Java Virtual Machine in a conventional implementation. This verification process consists of performing an abstract interpretation of the byte code that composes each method of an application. The aim is to statically check that the control flow and data flow do not generate errors (underflow or overflow of the stack, variable used with invalid type, ...). This algorithm is called *type inference* and its concepts were first introduced in [10]. The paper [11] gives an overview of the whole Java verification process. Java Card Virtual Machine (JCVM [3]) is a stack-based machine that uses particular registers named *local variables*. The Java specification [14] imposes some constraints on the byte code generated by compilers so that it can be verified. Because of these constraints, the type for each variable used by the program can be inferred. We use *stack map* to mean a structure giving the type of each variable, for each particular point of our program. Java programs are not linear; one can jump to a particular instruction from various sources. Thus we can have different hypotheses for the type of the variables; we need to *merge* all these hypothesis into one, and we need to find the corresponding type that matches all, *i.e.* a *compatible* one. This operation is called *unification* and is notated $\cap$. For example, for classes, it corresponds to the inheritance relation. We give, in the next paragraph, the standard version of the verification algorithm [14]. We also use this later on for the description of our implementation techniques for smart cards.

The standard algorithm uses the hypothesis that each instruction has an associated stack map, a TOS (Top of stack), and one *changed/unhanged* bit.

Initialization:

- mark first instruction as *changed*;
- fill its typing information by using the signature of the method (variables with no type are given the type $\top$ that means unusable variable), initialize the top of stack (TOS) at 0;
- for all other instructions:
  - mark them as *unchanged*,
  - initialise their TOS at -1 (this means that this instruction has not been checked earlier).

Main Loop:

- while there remain instructions marked as *changed*;
- choose an instruction I marked as *changed*;
- mark I as *unchanged*;
- simulate the execution of I over corresponding typing information (if there is an error, method is rejected);
- for each successor S of I:
  - we use O to mean the stack map for I and R to mean the stack map for S,
  - if S's TOS is equal to -1, then copy the stack map from O into R (we also initialize the TOS of I with the one of S),
  - otherwise, perform unification between each cells of the stack map for R and the ones for O. Result is denoted by $R \cap O$ (if the TOS does not correspond, verification stops with an error),
  - if $R \cap O \neq R$, mark S as *changed*; $R \cap O$ is now the new stack map for S.

3 Java Cards and verifiers

The standard verification algorithm is usually presented as being impossible to embed in a smart card, due to hardware limitations. Instead, the verification process is simplified and adapted in order to fit on small devices. In the next parts we present existing solutions that guarantee a good security level in a standard Java Card.

In a regular Java architecture, the produced application is verified while being loaded in the virtual machine. The byte-code verifier is define by [14] JVM specification as the basic component of safety and security in a standard JVM. Currently, SUN Java Card is built upon a split VM scheme: one part is called the "off-card VM"; the other one is called the "on-card VM". As the on-card VM cannot load a CLASS file format because of its complexity, a converter is used to produce CAP files (Converted APplet) that is more convenient to smart card constraints (no runtime linking, easy to link on load, ready to use class descriptor [20], ready to run byte code). While converting the class file, verification is performed in order to ensure that the produced CAP file is coherent with the CAP file structure definition. Smart cards are considered secure devices, and because of the lack of on-card verification it is currently impossible to download a new application to a card after it has been issued to an end-user. Some solutions have been proposed in order to achieve a good security level of the Java Card without using a verifier with regard to smart card constraints.

- **Digital signature**. The application designer sends the CAP file to a trusted third party that digitally signs it; hence, the card can now easily check if the application has been modified. This model guarantees a high level of security; the only requirement is to dispose of cryptographic routines on board for checking the validity of the signature. As smart cards already have a cryptographic co-processor, the verification cost is low. However, there is one major problem: it is a centralised deployment scheme. This centralization decreases the flexibility of such an approach: all cards need to be declared to the trusted third party. Moreover it is not compatible with a smart card that operates off line, nor to a massive smart card distribution (wolrdwide, there are roughly 1000 times more smart cards than PCs with web browsers).

- **Defensive VM**. As SUN has defined the conditions and rules required for executing each byte code of the Java Card language, a defensive virtual machine [4] can be used. Before executing a byte code, the virtual machine can perform all the required tests. Thus, a very high level of security is achieved, but the efficiency of the virtual machine is decreased, and the amount of working memory needed for running the applet is also increased significantly.

- **Proof-Carrying Code**. PCC techniques were introduced by G. Necula and P. Lee [17]. E. and K. Rose have adapted it to Java [18]. G. Necula and P. Lee have proposed an architecture to use PCC verification on a Java platform [5, 16]. A PCC verifier was developped by G. Grimaud [7] for a specialized language with regard to smart card constraints. An off-card part produces a proof (or certificate) that is joined to the application. The on-card verifier has just to perform a linear verification phase in order to check if the code is malicious. This verification is simple enough to be performed by the card. This is the solution that is recommended by SUN when implementing a KVM [19]. The size of the downloaded application is increased because of its proof (from 10% to 30%). The major problem is that the application deployment is now more complex because of the need for the proof generator. Valid applications can also be rejected only because their proof is not provided. Such a Java Card verifier as been fully embedded in a smart card by Gemplus Research Lab. It was developed using formal method and is described in [2].

- **Code transformation**. X. Leroy [21] has proposed another solution that is described in [12, 13]. It consists of some off-card code transformations (also named "normalization"), in such a way that each variable manipulated by the virtual machine has one and only one type during all the different execution paths of the program. The embedded part just needs to ensure that the code respects this property. Valid applications can also be rejected only because they were not

transformed using Trusted Logic[21] normalizer. The major problem is that performing such code transformations would increase the number of variables and also decrease the re-usability of them. Thus the card would need more memory resources for executing the programs, and it might refuse fully optimized code.

- **An infeasible solution (?): stand-alone verification**. Each of the solutions described earlier has some disadvantages that a stand-alone verifier would not have. The major problem is the need of some external pre processing of applets that can make a non stand-alone verifier refuse some valid applets. But the main stand-alone verification problem is its time and memory complexity.

Table 1 summarizes advantages and drawbacks of each solution.

In the next part we give some information about the hardware of smart cards, in order to explain the difficulties of having a stand-alone verifier on board.

4 Smart card constraints

Smart card has some hardware limitations due to ISO [9] conbstraints that are mainly defined to enforce smart card tamper resistance. Now we are going to focus on each of the hardware features of the smart card, starting with the microprocessor, then the memory and ending with the I/O.

- **Microprocessors**. A wide class of microprocessors are used from old 8-bit CISC micro chip (4.44 Mhz) to powerful 32-bit RISC (100 to 200 Mhz). The type of CPU used for smart card is highly influenced by the ISO[9] constraints linked to the card. For example, as the card is a portable device that is stored in a wallet, it must meet standards related to torsion and bending capacity. Table 2 gives an overview of some processors commonly used in smart cards. Historically, smart card manufacturers used 8-bit processors because operating systems and applications code are more compact. But smart cards now need to be more efficient and new embedded applications require more and more computing power. So card designers now choose 32-bit RISC processors (or 8/16 bits CISC). For our experimental prototype presented in the last part of this article, we are using an AVR from Atmel [1]. It is an 8/16 bit processor with 32 (8 bits) registers. Some of them can be grouped to perform computation on 16 bits values. Its is a typical platform in the smart card community for operating system design [15].

  Computing performance of a smart card is not a significant problem for a stand-along verifier. What is really the limiting factor for smart card programs is the very small amount of memory, and some annoying hardware-specific problems.

- **Memories**. Different types of memory exist on the card. The first one is the RAM (Random Access Memory); there is also some ROM (Read Only Memory); and finally EEPROM (Electric Erasable Programmable Read Only Memory) or FLASHRAM that are writable persistent memories. Because smart card silicon is suposed to be limited to 20 mm^2, the physical space needed for storing 1 bit is an important factor. Each kind of memories used is of different size concerning this point; the smallest is the ROM. Table 3 gives an overview of the amount of memory present on board, and also present the "memory cell" which is the size for 1 byte of memory on the micro module.

  Persistent memory has a major drawback, linked to its electronic properties. Its writing delay is up to 10000 times slower than RAM one. Furthermore, writing in persistent memory may damage the memory cells (the stress problem occurs when using the *erase* operation: making a bit going from 0 to 1). These constraints have led others developing light-weight byte-code verifiers to consider persistent memory only as a memory to store data and not has a working memory.

  EEPROM provides 4 primitive operations:

  - *read*: reading a value,

Solution	*Pro*	*Cons*
Crypto	dedicated hardware	centralized solution
Defensive VM	easy to implement	important run time penality
PCC	no run time penality	code overload, non standard deployment scheme
Code transformation		
Stand alone	standard deployment scheme	impossible ?

Table 1: Summarize of each solution

Model	Architecture	Data BUS size	Registers	Frequency
68H05	CISC	8 bits	2 (8 bits)	4.77 Mhz
AVR	RISC/CISC	8 bits	32 (8/16 bits)	4.77 Mhz
ARM7TI	RISC	32 bits	16 (32 bits)	4.77 to 28.16 Mhz

Table 2: Characteristics of common smart card microprocessors.

- *erase*: changing some bits from 0 to 1,
- *write*: changing some bits from 1 to 0,
- *update*: an *erase* followed by a *write*.

Erase is a stressing operation, it can provoke a lapse of the memory cell. It is also 38% slower than a *write*. Table 4 illustrates this characteristic. This characteristic is also true for FLASHRAM.

Operation	time for writing a 64 bytes page
erase	$\simeq$ 2.77 ms
write	$\simeq$ 2 ms

Table 4: Differences between the two writing operations in a typical EEPROM usage.

Using this memory well is the key technological challenge for smart card developers.

- **I/O**. We have one half duplex serial line (from 9600 to 15700 baud in conventional smart cards). This rate means that 1 KB of data is transferred in less than 1 second. Compared to the number of CPU cycles available in 1 second, the I/O capacity reduces the viability of technical solutions that involve an additional data transmission.

5 A fully embedded Java Card byte code verifier?

The byte code verifier is said to be impossible to be implemented inside a smart card; the impossibility is due to its high algorithmic complexity and also to its large memory consumption. In this part, we focus on all these constraints, and give solutions for allowing its usage on board, overcoming the hardware limits described previously.

5.1 Time and space complexity

The elementary operation we are going to use is the interpretation for a byte code working on a simple variable. Complexity is limited by $D*S*J*V$. D is the depth of the type lattice. S is the number of instructions of the verified program. J is the maximum number of jump (branching instructions), and V is the number of variable used at most. In the worst case, all instructions are jumps ($S = J$). By analyzing all the different byte codes of Java Card we can find the one that manipulates the highest number of variables. Let c represent this number, thus $V = c * S$ in the worst case. We can also state that we need one instruction for creating a new type in our lattice, thus $D = S$. Finally time complexity is $O(S^4)$. Hence, type inference is a polynomial form of the number of analyzed instructions. Let us now evaluate the memory that is needed for performing a typing inference. We need the type information for each label (destination of a branch) of

Type	memory point	capacity	write time	page size
ROM	reference	32-128 KB	read only	1 byte
FlashRAM	x 2-3	16-64 KB	2.5ms	64 bytes
EEPROM	x 4	4-64 KB	4ms	2-64 bytes
RAM	x 20	128-4096 B	$\leq 0.2\mu s$	1 byte

Table 3: Card memory characteristics

our program. The size of each of them is the number of local variables plus the maximal stack size. We also need one more frame for the current one. Thus the memory we need is $T*(S+L)*(J+1)$ where J is the number of branching instructions and/or exception handlers, T the size needed to encode one type, S the stack size, and L the number of local variables. Practically, we can easily require more than 3 KB of RAM. Thus we need to store the proof in persistent memory. Doing this we need to be aware of the stressing problem and also of the slowness of writing of persistent memory.

5.2 Our solutions

We propose to use persistent memory (*i.e.* EEPROM or FLASHRAM) for storing stack maps of the current method. Nevertheless, the amount of memory needed is smaller than the one for a PCC verifier [2] that needs to store the proof for every methods in persistent memory. Our strategy consists of using RAM to hold the entire stack map whenever possible, and when not possible to use it as a cache for persistent memory. The first challenge is the bad property of persistent memory related to cells stressing. We propose the usage of a non stressing type encoding. The second challenge is to find the best cache policy according to our problem.

5.3 A non stressing type encoding

In order to solve the problem of persistent memory stress, we propose to find a good encoding of the type lattice used during type inference. The goal is to use a non-stressing write when storing a type in persistent memory. The paper [8] presents examples of type lattice encoding that are used in compilation or in databases. It proposes techniques to perform a bijection between lattice operations and Boolean operations. Described encodings allow finding the Least Upper Bound (LUB) of two elements using a composition of logical functions. Type unification consists of finding the LUB of two types. Typing information needs, sometimes, to be store in persistent memory due to its size, and persistent memory has a stressing problem. We observe that when unifying two types, the result type is higher in the type lattice, so we would like to be able to move upward in the lattice while only changing bits from 1 to 0. Such an encoding causes no stress on persistent memory, and unification is reduced to a logical AND that has the property of only clearing bits (1 to 0 transitions). As non stressing writes takes less time than a stressing one, unification goes faster.

We could find a more compact encoding (*i.e.* using less memory) by using a more complex Boolean function, but this would take less into account persistent memory properties.

In our prototype, dynamic downloaded classes do not take benefits of the encoding, as it would require computing the whole type encoding. However our experiments show that complex unification (*i.e.* one between two classes, producing a class that is not `java.lang.Object`) happens very rarely. In fact, smart card software engineering implies using specific rules that reduce the number of different class used at the same time. Accordingly, we do not try to optimize these cases, we just accept the minor performance degradation. This choice has no effect on security. Experimentally, we found that less than 1% of unification are performed between two classes. All these techniques reduce $\cap$ computation time and ensure a non stressing usage of persistent memory but they do not deal with the latency of write operation.

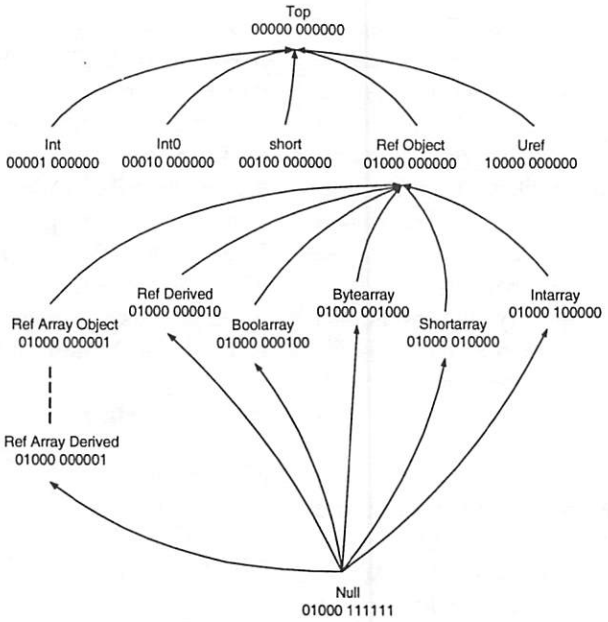


Figure 1: Our non stressing type encoding

5.4 Efficient fixed points using software cache policy

As described earlier, type inference is an iterative process that stops when a fixed point is reached. By using our type encoding we decrease the persistent memory stress, but we can also obtain better results by trying to maximize the RAM usage. The aim is to maximize the size of the data we can store and work on in a RAM. For example, some specific methods extracted from our application test pool need more than 4 KB of memory for the verification process. As RAM memory is faster than persistent memory, but is less present on a smart card, we use a *software cache* for working on typing information. Such a technique can speed up the verification of huge (13KB) applets as it highly decreases the number of write in persistent memory.

In order to perform type inference, we need to find the target of branching instructions. Having the branching instructions and also the branched ones, we can build (at low cost) the control graph flow of the verified program. Then, when verifying a particular code section we know which are the ones we can jump to by using the control graph flow. If we look at the example given in Figure 2, when verifying the last code sec-

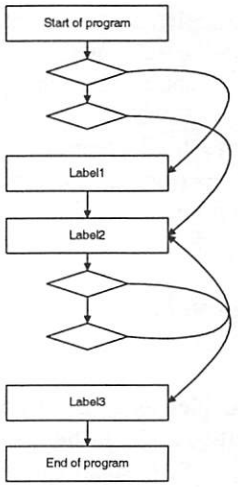


Figure 2: An example of a cache policy using control graph flow

tion (the one beginning with Label 3), we can see that none of the other code section are accessible. The control graph flow is used for piloting the eviction of data in the cache. Moreover, we can sometimes extract from the graph the property that a label is not reachable from any further point of the program. In this case, eviction performs no write in persistent memory as the type information will not be used anymore. Pratically, this happens frequently in conventional Java Card code. The eviction significantly increase the speed of stand-alone byte code verification.

We note that the techniques we propose are compatible with future types of persistent memory like FeRAM (Ferro-Electric Random Access Memory) or MRAM (Magnetic Random Access Memory), and will take benefits of using these memories. The techniques we have presented here have been delivered to the industry and are protected under the patent [6].

6 Experiments

We have implemented a prototype that includes each of the techniques we have mentioned. This prototype shows that the stand-alone byte code verifier is feasible in a smart card. Moreover, we have used common benchmarks for evaluating its performance against a system using a PCC verifier.

6.1 Our prototype

We have implemented the concepts and techniques described in the earlier part of this article on an AVR 3232SC smart card microprocessor from ATMEL. This chip includes 32 KB of persistent memory, 32 KB of ROM, and a maximum of 768 bytes of RAM. The amount of available RAM highly depends on the state of the verifier and also of the virtual machine. Practically, in most of cases, using less than 512 bytes of RAM does not alter stand-alone byte code verification's efficiency. The type inference algorithm has an important advantage: we can compute the amount of memory we will need to verify a particular method of an application. We propose different algorithms and heuristics: we have implemented different cache policies that suit a particular amount of RAM (an "all in RAM policy", a LRU policy, a more evolved LRU, and finally the one described earlier that uses control graph flow of the verified program). We also use some heuristics for selecting the current label to verify. These heuristics have different properties in term of performance. We choose a heuristic dynamically in order to fit the amount of working memory. We give in Table 5 the size of the important part of our verifier.

Module name	code size in bytes
Byte code transition rules	13552
Unification	2242
Cache policies	3700

Table 5: Memory footprint in ROM

We have a total of 30 KB of code for our stand-alone type verifier. The remaining part which is not described in table 5 consists of routines that deal with the cap file format, I/O, and RAM and EEPROM allocators. We did not tune this implementation for our proof of concept, and we would expect a production version to be both smaller and faster. For example, a standard JCVM represents 20KB of code; its API represents 60 to 100KB; and a huge user application is about 13KB (these measures are extracted from products from our industrial partner).

6.2 Practical metrics

We give in Table 6 the duration of the loading phase in milliseconds, and also of verifications, for three applets which size are from 3 KB to 4.5 KB. The PCC verifier is the one described in [2].

The loading phase for the PCC is higher than the one for the stand-alone (between 20% to 30%). It is due to the fact that it needs to load and write the entire proof in persistent memory. The overhead for the proof in a PCC verifier is said to be between 20 to 30 % in related work. The time taken to write the proof in persistent memory before verification by the PCC could be avoid by performing the verification on the fly. Thus the only difference between a PCC and a stand-alone verifier would be the extra loading

	Applet	PCC	Stand-alone	Stand-alone Vs PCC
Load	grl-com.utils.wallet	3335	2744	0.82
	grl-com.games.tictactoe	9594	8933	0.93
	com.gemplus.pacap.utils	9414	7331	0.78
Verif	grl-com.utils.wallet	411	318	0.77
	grl-com.games.tictactoe	1232	1102	0.89
	com.gemplus.pacap.utils	1312	1463	1.12
Total	grl-com.utils.wallet	3746	3062	0.82
	grl-com.games.tictactoe	10826	10035	0.92
	com.gemplus.pacap.utils	10726	8794	0.82

Table 6: Time for loading and verifying (in ms)

time of the proof. Of course, the techniques described earlier could be used on the PCC verifier to reduce the time needed to use the proof (on the fly verification, non stressing encoding to speed up operations on types, ...). We need also to remind that the off-card generation of the proof for a PCC verifier as a cost in terms of deployment tools and also in terms of time. Thus PCC as a model is more expensive than a stand-alone verifier.

If we look at the global time taken for verification, we can point out that stand-alone verification is not slower than PCC one which has a linear complexity. In some particular case, stand-alone verification can be faster than PCC one. Smart card application are often simple in terms of generated byte code. Thus stand-alone verification is nearly linear. With this table we show that the extra time needed for stand-alone verification (with our technical approach) is often less than those needed for downloading and storing the PCC proof.

Some experimental results of a verifier using code transformation were presented in [13] but with only few details. Nevertheless, we are in the same order of magnitude for verification execution time (1 sec for 1 kb of code). We did not have access to smart card using others verifications strategies so we could not compare them with our prototype. But cryptographic signature supposes an extra time to check the basic cap file signature after download (for example, a smart card usually performs a RSA in something like 10ms per 64 bytes). Concerning the solution using digital signature, it is the worst solution in term of infrastructure as it assumes a trusted third party to sign applications.

Conclusion

We have shown that careful attention to the smart card hardware allows us to integrate a stand-alone verifier on a smart card. Stand-alone verification is no longer a mismatch to the smart card constraints. The usage of hardware-specific techniques allows the stand-alone verifier to be as efficient as a PCC one.

References

[1] Atmel Corporation. Atmel AVR. `http://www.atmel.com`.

[2] L. Casset, L. Burdy, and A. Requet. Formal development of an embedded verifier for java card byte code. In *DSN-2002. The International Conference on Dependable Systems and Networks*, 2002.

[3] Z. Chen. *Java Card Technology for Smart Cards*. Addison Wesley, 2000.

[4] R. M. Cohen. Guide to the djvm model version 0.5 alpha ** draft **, 1997.

[5] C. Colby, G. C. Necula, and P. Lee. A Proof-Carrying Code Architecture for Java. In *Computer Aided Verification*, 2000.

[6] D. Deville, G. Grimaud, and A. Requet. Efficient representation of code verifier structures, 2001. International pending patent.

[7] G. Grimaud, J. L. Lanet, and J. J. Vandewalle. Façade: A typed intermediate language dedicated to smart card. In *Software*

Engineering - ESEC/FSE'99, pages 476–493, 1999.

[8] H. Ait-Kaci and R. Boyer and P. Lincoln and R. Nasr. Efficient Implementation of Lattice Operations. *ACM Transactions on Programming Languages and Systems, TOPLAS*, 11(1):115–146, 1989.

[9] International Standard Organisation: ISO. Integrated circuit(s) cards with contacts, parts 1 to 9, 1987-1998.

[10] Gary A. Kildall. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[11] X. Leroy. Java bytecode verification : an overview. In *Computer Aided Verification*, 2001.

[12] X. Leroy. On-card bytecode verification for java card. In *Esmart*, 2001.

[13] X. Leroy. Bytecode verification for Java smart card. *Software Practice & Experience*, 32:319–340, 2002.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[15] Matthias Bruestle. SOSSE - Simple Operating System for Smartcard Education, 2002. `http://www.franken.de/users/mbsks/sosse/html/`.

[16] G. C. Necula. A scalable architecture for proof-carrying-code. In *Fifth International Symposium of Functionnal and Logic Programming*, 2001.

[17] G. C. Necula and P. Lee. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, Paris, January 1997.

[18] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop of the Formal Underpinnings of the Java Paradigm, OOPSLA'98*, 1998.

[19] Sun Microsystem. Connected Limited Device Configuration and K Virtual Machine. `http://java.sun.com/products/cldc/`.

[20] Sun Microsystem. The javacard™ 2.1 specification. `http://java.sun.com/products/javacard/`.

[21] Trusted Logic. Formal methods, smart card, security. `http://www.trustedlogic.com/`.

Enhancements for Hyper-Threading Technology in the Operating System – Seeking the Optimal Scheduling

Jun Nakajima
Software and Solutions Group, Intel Corporation
Venkatesh Pallipadi
Software and Solutions Group, Intel Corporation

Abstract

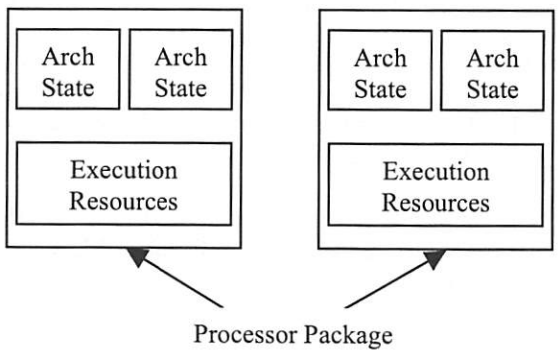
Hyper-Threading Technology (HT) is Intel®'s implementation of Simultaneous Multi-Threading (SMT). HT delivers two logical processors that can execute different tasks simultaneously using shared hardware resources on the processor package. In general a multi-threaded application that scales well and is optimized on SMP systems should be able to benefit on systems with HT as well for most cases, without any changes, although the operating system (OS) needs HT-specific enhancements. Among those we found process scheduling is one of the most crucial enhancements required in the OS, and we have been seeking the optimal scheduling for HT, evaluating various ideas and algorithms. One of our finds is, to efficiently utilize such execution resources, severe resource contention against the same and limited execution resource should be avoided in a processor package. The OS can attempt to utilize the CPU execution resources in processor packages if it can monitor and predict how the processes and system utilize the CPU execution resources in the multiprocessor environment. We have implemented a supplementary functionality for the scheduler called "Micro-Architectural Scheduling Assist (MASA)" on Linux (2.4.18) chiefly as a user program, rather than in the scheduler itself. This is because we believe that the users can tune the system more effectively for various workloads if we clarify how it works as a distinct entity. Most of this problem and solution is generic; all we require is for the OS to support an API for processor affinity and to provide per-thread or per-process hardware event counts from the hardware performance monitoring counters at runtime.

1. Introduction

1.1. Overview of HT

Hyper-Threading Technology (HT) is Intel®'s implementation of Simultaneous Multi-Threading (SMT) ([9],[10]). SMT machines increase utilization of the execution resources in a processor package and speedup the execution of jobs, by fetching and executing multiple instruction streams.

Figure 1: Architecture state and shared execution resources function as a logical processor.



By utilizing the process-level or thread-level parallelisms in applications, twice as many (hardware) threads can be dispatched and executing concurrently using the execution resources in a processor package. Each logical processor maintains a complete set of the architecture state (See Figure 1). The architecture state consists of registers, which includes the general-purpose registers, the control registers, the advanced programmable interrupt controller (APIC) registers and some machine state registers. From the software perspective, once the architecture state is duplicated, the processor appears to be two processors.

One logical processor can utilize excess resource bandwidth that is not consumed by the other logical processor, allowing the other task to make progress. This way, both the overall utilization of execution resources as well as the overall performance of software applications in the multi-tasking/multi-threading environment increases. Logical processors share nearly all other resources on the physical processor, such as caches, execution units, branch predictors, control logic, and buses.

HT is not expected to make a given single-threaded application to execute faster when executing alone, but when two or more unrelated applications are exe-

cuting under HT, the overall system throughput can improve due to HT. See [5] for details.

1.2. General HT enhancements in the Operating System

This section describes the typical enhancements, to explain the implications of HT to the OS. Following is a summary of enhancements recommended in the OS.

Detection of HT – The OS needs to detect both the logical and processor packages if HT is available for that processor(s).

`hlt` at idle loop – The IA-32 Intel® Architecture has an instruction call hlt (halt) that stops processor execution and normally allows the processor to go into a lower-power mode. On a processor with HT, executing `hlt` transitions from a multi-task mode to a single-task mode, giving the other logical processor full use of all processor execution resources; see [5] for the details.

`pause` instruction at spin-waits – The OS typically uses synchronization primitives, such as spin locks in multiprocessor systems. The `pause` is equivalent to "`rep;nop`" for all known Intel® architecture prior to Pentium® 4 or Intel® Xeon™ processors. The instruction in spin-waits can avoid severe penalty generated when a processor is spinning on a synchronization variable at full speed.

Special handling for shared physical resources – MTRRs (Memory Type Range Registers) and the microcode are shared by the logical processors on a processor package. The OS needs to ensure the update to those registers is synchronized between the logical processors and it happens just once per processor package, as opposed to once per logical processor, if required by the spec.

Preventing excessive eviction in first-level data cache – Cached data in the first-level data cache are tagged and indexed by virtual addresses. This means two processes running on a different logical processors on a processor package can cause repeated evictions and allocations of cache lines when they are accessing the same virtual address or near in a competing fashion (e.g. user stack).

The original Linux kernel, for example, sets the same value to the initial user stack pointer in every user process. In our enhancement, we offset the stack pointer simply by a multiple of 128 bytes using the mod 64, i.e. ((pid%64) << 7) of the unique process ID to resolve this issue.

Scalability issues – The current Linux, for example, is scalable in most cases, at least up to 8 CPUs. However, enabling HT means doubling the number of processors in the system, thus it can expose scalability issues, or it does not show performance enhancements when HT is enabled.

Linux (2.4.17 or higher) supports HT, and it has all the above changes in it. We developed and identified the essential code (about just 1000 lines code) for those changes (except scalability issues) based on performance measurements, and then improved the code with Linux community.

APPENDIX locates the relevant lines for the changes in Linux 2.4.19 (the latest tree as of writing). Those changes or requirements should be applicable to an OS in general when supporting HT, although some of them might need to be re-implemented for the target OS.

1.3. Basic scheduler optimizations for HT

This section describes the basic scheduler related enhancements for HT.

Processor-cache affinity – Processor-cache affinity is a commonly used technique in modern operating systems; see [8] for example, for the benefits of exploiting processor-cache affinity information in scheduling decisions. It is more effective on the HT systems in the sense that a process can benefit from processor-cache affinity even if moved to the other logical processor within a processor package.

Since the L2 cache is shared in a processor package, however, the hit (or miss) ratio can depend on how the other logical processor uses the L2 cache as well. If the current process on a processor consumes the L2 cache substantially, it can affect the processes running on the other logical processors. Therefore, to avoid performance degradation caused by cache thrashing between the two logical processors, we need to monitor and minimize such L2 cache misses in a processor package.

Note that excessive L2 cache misses also can affect the entire system, causing significant traffic on the system (front-side) bus.

Page coloring (for example, see [4]) could reduce occurrence of severe impacts on two different processes or threads caused by competitive eviction of L2 cache lines in a processor package. If two different processes access their own data very frequently, and the pages associated with the data happen to have the same color, the possibility of competitive eviction of L2 cache lines can be higher, compared to the case where page coloring is implemented. The same discussion is applicable to the threads in a multi-threaded application.

Although there are some patches are available for page coloring in Linux, we haven't measured the benefits of page coloring for HT.

HT-Aware idle handling – This enhancement in the scheduler significantly improves performance when the load is relatively low. For the scheduling purposes, the OS needs to find idle CPUs, if any. On HT systems, however, the processor package is not necessarily idle even if one of the logical processor is idle; the other may be very active. Therefore, the scheduler needs to prioritize "real-idle" (both logical processors are idle) over "half-idle" (one of them is idle), when dispatching a process to a logical processor to obtain higher performance.

This attribute also helps to avoid the situation where two processes run on a processor package but the other package is completely idle in a 2-way SMP system. However, this kind of situation cannot always be prevented because the OS cannot predict when a particular process terminates. Once this situation occurs, the scheduler usually does not resolve it.

Scalability of the scheduler – The Linux original uses a single global run queue with a global spin lock. This scheduler works well for most cases, but there are some scalability issues especially handling a large number of processes/threads. The O(1) scheduler from Ingo Molnar is proposed to resolve such issues, and it uses per-CPU run queue and a spin lock for each, and locking is not required as long as the CPU manipulates its own run queue.

2. Related works

We discuss how the existing techniques can contribute to performance enhancements of HT systems. In this paper, we assume that processes are scheduled in a time-sharing fashion.

2.1. Performance Monitoring Counter

The work [1] is interesting in that it combined the hardware monitoring counters and program-centric code annotations to guide thread scheduling on SMP, to improve thread locality. Some findings show that some workloads achieved speedup almost entirely through user annotations, and for some long-lived ones speedup is gained by preserving locality with each thread.

We need to run a process for some time, to get the information of its workload. Such user's annotation (including processor binding) would be helpful.

We use hardware performance monitoring counters (simply performance monitoring counter, hereafter) to get such micro-architectural information. See [1] for general and detailed description and benefits of performance monitoring counters that are available on various architectures. The major benefit of using performance monitoring counters is to allow software to obtain detailed and specific information at runtime without impacting the performance. Usually, performance monitoring counters are used to tune applications and the OS. There are some tuning tools that use them for the Intel® architectures, such as VTune [3], for example.

Performance monitoring counters can be micro-architecture specific, not architecture-specific. This means, the performance monitoring counters available can vary even if the architecture appears same. In terms of the OS implementation, this is an issue, and we resolve it by defining load metric, rather than bare performance monitoring counters. We discuss it later in Section 4.1.

We don't see conflicts with such tools and the scheduler, although the number of performance monitoring counters available will be reduced. Since performance monitoring counters are the system-wide resources, the OS should provide API for allocating the counters to avoid conflicts among such tools or drivers.

2.2. Symbiotic Jobscheduling

We share the issues to resolve with the symbiotic jobscheduler ([6], [7]):

- Jobs in an SMT processor can conflict with each other on various shared system resources.

- The scheduling has to be aware of the SMT requirements to find the optimized utilization of execution resources.

However, the target system and the methodology is quite different:

- We don't attempt to make a running set of jobs that will be coscheduled, by discovering efficient schedules in a processor on the fly. Instead, we attempt to detect interference or conflicts with the execution resources in SMP systems consisting of multiple SMT, i.e. HT processors, and to balance the load among such processors. Therefore, uni-processor (UP) systems are not our target. We believe the programmer/developer can tune their application better in the UP case using proper performance tuning tools.

- We don't require modifications to the OS scheduler in the kernel that needs to support SMP systems as well. Inventing a new scheduler only for SMT can generate maintenance and QA issues in a commercial OS. Instead, we provided a user-mode daemon that monitors hardware events in the processors, to detect such conflicts with the execution resources.

2.3. Load balancing

In the multiprocessor environment, a typical load metric employed by the OS scheduler at load balancing time is the length of the run queue(s). This is generic and effective when keeping fairness of the processes that are run in a time-sharing fashion, because the processes on a processor with a shorter run queue can have more chances to run.

At the same time the length of the run queue does not reflect the load of the processor or processor package, because the workload or the execution resources utilized at runtime can vary from process to process.

3. Advanced HT Optimizations in the Scheduler

In this section we discuss advanced HT optimizations in the scheduler. As such, we are evaluating their effectiveness gathering data from various workloads.

3.1. Motivation

When HT is enabled, the number of the processors looks doubled to the OS, because each of them (logical processor) has architectural state as a processor, and BIOS reports as such. In general a multi-threaded application that scales well and is optimized on SMP systems should be able to benefit on systems with HT.

In the multiprocessor environment, however, there can be process placement issues, because the two logical processors in a processor package share the execution resources.

***Note** - For the following illustrations we have used **164.gzip of SPEC-CPU2000** (referred as **INT**) as the integer operation intensive benchmark, **177.mesa of SPEC-CPU2000** (referred as **FP**) as the floating-point intensive benchmark and **197.parser of SPEC-CPU2000** (referred as **L2C**) as the L2 cache intensive benchmark. See [12] for details.*

The floating-point execution unit, for example, is one of the limited execution resources in a processor package. If we run two processes of an integer-intensive program (INT) and two of floating-point intensive program (FP) on a 2-way (4way HT) machine, binding them to the processor package, we see significant performance difference depending process placement (see Table 1).

Another example of a critical execution resource is L2 cache unit. If we run two processes of an integer-intensive program (INT) and two of a program (L2C) that consumes L2 cache lines intensively, we also see visible performance difference depending on process placement (see Table 2).

To explain the benefits of HT, we also run the test case with HT disabled as well. The "(" and ")" indicate a processor package with HT *enabled*, and "[" and "]" a physical processor with HT *disabled*. "(INT, INT) (FP, FP)", for example, means:

- Two INT programs are bound to a processor package, and two FP programs are bound to the other processor package. HT is enabled, and the system looks a 4-way SMP.

"[INT, INT] [FP, FP]" means,

- Two INT programs are bound to a processor, and two FP programs are bound to the other processor. HT is disabled, and the system looks a dual-processor system. The two programs are not run at the same time in a processor, but in a TSS fashion.

We made the measurements on a system that has dual Intel® Xeon™.

Placement	INT	INT	FP	FP	Total
(INT, FP) (INT, FP)	269.3	270.7	234.0	234.7	1008.7
(INT, INT) (FP, FP)	317.6	316.0	256.3	255.7	1145.6
[INT, FP] [INT, FP]	342.0	337.6	271.6	272.3	1223.6
[INT, INT] [FP, FP]	343.0	342.6	268.7	268.3	1222.6

Table 1: Average time (sec.) to complete (INT, FP)

Placement	INT	INT	L2C	L2C	Total
(INT, L2C) (INT, L2C)	291.0	291.0	437.7	440.3	1460.0
(INT, INT) (L2C, L2C)	317.3	317.7	462.0	459.0	1556.0
[INT, L2C] [INT, L2C]	342.6	341.7	509.0	509.7	1703.0
[INT, INT] [L2C, L2C]	344.3	345.7	492.3	492.0	1674.3

Table 2: Average time (sec.) to complete (INT, L2C)

"Total Time" means the sum of the average time (in second) to complete each program (4 of them). As we in both Table 1 and Table 2,

- Performance is always better when HT is enabled (about 17% in Table 1 and 13% in Table 2). This is because HT is utilizing the execution resources more by running the two logical processors in a processor packages.

- For HT, combining different workloads is better (about 12% in Table 1 and 6% in Table 2). This is because such different workloads tend to have less interference with the execution resources.

Table 3 shows values in the hardware performance monitoring counters for the cases in Table 1 and Table 2. The samples were taken in a 5-second period when the test cases were run. Table 4 lists the placements in the Table 1, Table 2, and Base (HT enabled) as well, where a single instance of FP or L2C runs at full-speed with the other processor package idle. For each, it includes the number of L2 cache misses, floating-point uops (decoded IA-32 floating-point instructions) retired (actually executed), and instructions retired. CPU/Package means the logical CPU ID and the processor package that it belongs to. Note that the process "masad" is our user-mode micro-architectural scheduling assist daemon, and we used it to control placement of the INT and FP or L2C processes. Also note that the execution resources used by the process is negligible.

Table 4 summarizes Table 3 per processor package, providing the number of the L2 cache misses, FP uops retired, and the ratios for those events per instruction retired (processor package 0 or 1 in Base was omitted because only a single instance of the program was run.). Notice the following things for the FP test case:

- The total instructions retired are more in (INT, FP) (INT, FP) than the others. This implies this setup achieves higher overall throughput than the other cases.

- The number of FP uops in (INT, INT) (FP, FP) is outstanding in the processor package 1. And it is more than the one in Base.

The above observation suggests that FP execution resources are causing over-subscription or interference between the FP processes.

The above discussion applies to the L2C test as well. See Table 3 and Table 4:

- One of the processor packages has a higher number of the total instructions retired, but the other one has significantly lower (5.815E+09) with (INT, INT) (L2C, L2C).

- The number of L2 cache misses is outstanding in the processor package 1 with (INT, INT) (L2C, L2C).

The above observation again suggests that L2C execution resources are causing over-subscription or interference between the L2C processes.

The problem above happens with practical multi-threaded applications, and in fact it is a generic SMP issue as well. Inktomi Traffic Server, for example, can be tuned by setting CPU affinity. Performance is measured on a 2-way SMP (4-way with HT) machine by request rate (req/sec) within 1600 ms response time. Since the threads in the application can run any processor, they occasionally run on the same processor package, and showing worse (unacceptable) re-

sponse time (1638.08 ms). In many cases, a typical technique is to bind the processes or threads of interest to CPUs. In case of HT, we need to bind such processes or threads to a processor package or a set of logical processors. If we set CPU affinity, binding the thread to each processor package, we see better performance with shorter response time (1436.12 ms).

Placement	CPU/Package	Process	L2 cache miss	FP uops	Inst. Retired
(INT, FP) (INT, FP)	2/0	init	73	0	1955
	2/0	masad	152	1389	60275
	2/0	INT	2660214	0	5531725966
	0/0	FP	1203806	61710721	5073047685
	3/1	sendmail	116	16	6737
	3/1	emacs	7430	29	91376
	3/1	INT	2641327	1	5550636901
	1/1	FP	1946383	61444159	4859221730
(INT, INT) (FP, FP)	2/0	masad	181	1389	62002
	2/0	INT	4604056	61	4812888475
	0/0	INT	5219503	105	4780357278
	3/1	FP	1434903	62467839	4674550450
	1/1	init	70	0	1955
	1/1	sendmail	115	14	6759
	1/1	sshd	259	107	118998
	1/1	emacs	7875	3649	345118
	1/1	FP	1355739	60287239	4619649049
Base	2/0	init	69	0	1955
	0/0	FP	847987	106756347	8713126869
	3/1	emacs	1965	2702	350680
	3/1	sendmail	76	16	6666
	3/1	masad	108	820	41724
	1/1	sshd	55	64	83267

Placement	CPU/Package	Process	L2 cache miss	FP uops	Inst. Retired
(INT, L2C) (INT, L2C)	2/0	init	85	0	1955
	2/0	masad	1445	1445	66739
	2/0	L2C	34197735	143644	3420942349
	0/0	INT	4832876	0	4972729362
	2/1	sendmail	105	16	6759
	1/1	emacs	826	2733	355311
	1/1	INT	10127606	150	4548505455
	3/1	L2C	38214884	580769	3744309183
(INT, INT) (L2C, L2C)	2/0	masad	159	1379	66105
	2/0	INT	5269351	127	4726320225
	2/0	init	85	0	1955
	2/0	sendmail	112	16	6759
	2/0	sshd	179	26	90156
	0/0	INT	4600986	117	5404163660
	3/1	L2C	39424290	55080	2627281901
	1/1	emacs	9679	4776	815817
	1/1	L2C	34522937	81583	3186873833
Base	0/0	sshd	57	60	90212
	0/0	emacs	1965	2702	350680
	1/1	L2C	46925899	1246614	6577434630
	3/1	init	57	0	1955
	3/1	sendmail	115	16	6666
	3/1	masad	147	822	42455

Table 3: Performance Counter Values

Affinity Set	Request Rate (req/sec.)	Response Time (ms)
Off	1600	1638.38
On	1700	1436.12

Table 5: Inktomi Traffic Server Performance Difference

It is true that the same problem occurs with in the SMP environment as well, but resolving the issue is more difficult in the HT environment. In the HT environment, each of the net threads is actually running on a logical processor in a processor package. *And we need to migrate the currently running one to another logical processor in the other processor package (i.e. process migration), rather than one in the run queue as in the SMP case.* In the SMP environment, even if the two instances of the net thread are in the same CPU, at most one is running and the other one must be on the run queue. In other words, a scheduler with a proper load balance mechanism should be able to handle this case relatively easily in the SMP environment.

We have used limited number of processes for the illustrations here. However, this kind of problem can occur in heavily loaded system too, if the currently running processes on two logical processors of a package use similar kind of execution resources.

To understand and resolve those issues, we need to look at the micro-architecture level, because the amount of execution resources and dependencies are micro-architecture specific. In other words, the severity of the problem above can vary from micro-architecture to micro-architecture, even though the architecture implemented is equivalent. In addition, it's barely possible to tell which and how execution resources are used to execute a given instruction, as long as we are looking at the instruction/architecture level.

3.2. Manual Binding of Processes

Binding a process/thread to a CPU(s) is commonly supported on a typical OS, including Linux.

In Linux, the following API are available in branch trees (2.5 and Alan Cox's tree as of today), and will be available soon in the base (i.e. 2.4 tree):

- *sched_setaffinity*(pid, len, *cpu_mask) – set affinity.
- *sched_getaffinity*(pid, len, *cpu_mask) – get affinity.

The first argument *pid* is the process ID of the target process, and the third argument *cpu_mask* specifies the bit set of the CPUs to bind the process to. The second argument *len* specifies the length of the CPU mask. For example, to bind a process with the process ID 46732, to the logical CPU 0 and 2, one can use the following C code:

```
unsigned long cpu_mask = 0x5;

sched_setaffinity(46732,
     sizeof (unsigned long), &cpu_mask);
```

As we present in 3.1, this method is effective for HT systems as well, as long as the system behaves as the user expects and understands. However, this is fragile in the following aspects:

- The optimal process/processor binding can change, as the system configuration (e.g. the

Placement	Package	L2C miss	L2C miss/Inst.	FP uops	FP uops/Inst	Total Inst. Retired
(INT, FP) (INT, FP)	0	3.864E+06	3.644E-04	6.171E+07	5.819E-03	1.060E+10
	1	4.595E+06	4.414E-04	6.144E+07	5.902E-03	1.041E+10
(INT, INT) (FP, FP)	0	9.824E+06	1.024E-03	1.555E+03	1.621E-07	9.593E+09
	1	2.799E+06	3.011E-04	1.228E+08	1.321E-02	9.295E+09
Base	0	8.481E+05	9.733E-05	1.068E+08	1.225E-02	8.713E+09

Placement	Package	L2C miss	L2C miss/Inst.	FP uops	FP uops/Inst	Total Inst. Retired
(INT, L2C) (INT, L2C)	0	3.903E+07	4.650E-03	1.451E+05	1.729E-05	8.394E+09
	1	4.834E+07	5.829E-03	5.850E+05	7.054E-05	8.293E+09
(INT, INT) (LC2, L2C)	0	9.871E+06	9.744E-04	1.665E+03	1.644E-07	1.013E+10
	1	7.396E+07	1.272E-02	1.414E+05	2.432E-05	5.815E+09
Base	1	4.693E+07	7.134E-03	1.247E+06	1.895E-04	6.577E+09

Table 4: Execution Resource Usage

number/speed of processors, memory installed, I/O subsystems) changes.

- The user sometimes has difficulty identifying the optimal placement as long as he or she looks at the instruction level, rather than the micro-architecture level.

- The scope of a particular application is static and local, and the assumption can be wrong when the other system activities, such as interrupts or swap handling, are active at runtime.

- The micro-architecture can change, and the optimal process/processor binding can also change because of that. If the user needs to run the same system on processors with a newer or different micro-architecture, the system may need different process/processor binding. The *load calibration* technique discussed in Section 4.1 attempts to solve this problem.

These issues are common between SMT and MP systems (including MP systems of SMT processors), the impacts caused by a wrong setup or tuning could be more exposed in SMT, mainly because the hardware threads (i.e. logical processors in the OS) are not necessarily physically independent. If the system in Section 3.1 were a usual 4-way SMP machine, the total would be same for any process placement in the both Table 1 and Table 2.

3.3. Changes in Kernel or User

The problems above can happen especially when a process/thread is bound to a CPU(s) statically. If we can adjust such affinity at runtime, i.e. reset affinity accordingly, we can avoid such problems. This motivated us to employ the micro-architectural information or hardware performance monitoring counters at runtime when adjusting or tuning placement of the processes on the processors/processor packages. Since we know static binding can provide fairly reasonable performance, we don't believe we need to change affinity very frequently. Rather, we attempt to detect wrong placement that causes performance degradation, and to resolve such a situation.

The other benefit of this approach is that we can provide such a load balancing policy as a user-level program if:

- Getting/setting of CPU binding is available for either a process or software thread, depending on the unit scheduled by the OS, and

- Per-process or per-thread hardware event counts from hardware performance monitoring counters are available. A device driver can provide this information. Some OS maintains the threads in a process in the kernel whereas a thread in Linux, for example, is implemented as a process and the kernel has no idea about the mapping between threads and processes.

The benefits of having user-level policy are significant:

- Extending the load-balancing algorithm in the OS does not solve the case where two very active processes are "stuck" in a processor package, because it does not migrate the currently running processes.

- The users can tune the system more effectively for various workloads if we clarify how it works as a distinct entity, rather than providing a black box in the scheduler.

- The scheduler in the OS can be architecture independent especially when it needs to support multiple architectures. Thus it is not good idea to change it for a particular architecture for various reasons (performance impacts, verification, maintenance, etc.)

- The kernel usually does not use floating-point operations except for very limited case. Such a program would need to process a number of large values (from performance monitoring counters). It would be efficiently written with floating-point operations available.

The next section explains our proposal to do this optimization using a user-mode monitoring daemon.

4. Micro-Architectural Scheduling Assist (MASA) Methodology

As we discussed our requirements for the OS in 3.3, the Micro-Architectural Scheduling Assist (MASA) should be available if the OS provides the generic requirements. In the following section, we discuss the methodology.

Since the micro-architecture implements the architecture of the processor, it can vary even for a same processor family in details, such as the numbers of execution units, or size of L1/L2 caches. Since our purpose to monitor utilization of limited and shared

execution resources for HT, we define load metrics to evaluate utilization of those resources, based chiefly on the Intel® Xeon™ processor family.

Load Metric

A load metric is an abstracted performance monitoring counter in a sense, and it is defined as a formula using a set of performance-monitoring event counters, focusing on events on a particular micro-architectural resource(s). Since the events monitored by the performance monitoring counters are typically very detailed and specific, it is comprehensive and efficient if we combine them as a functional unit. In addition, we can reduce the differences among the performance monitoring counters that can be micro-architecture specific, but can vary from architecture to architecture. A performance monitoring counter is incremented when an event specified occurs. We define the following load metrics based on the micro-architecture of the Intel® Pentum® 4 or Intel® Xeon™ Processors [2]:

- **The number FP operations executed.** This represents load against FP Move and FP Execute unit.
- **The number of cache and TLB related activities.** L1/L2 cache misses, TLB misses. We are interested in more in the local impacts, rather than the system-wide ones.
- **The number of memory load/store operations.** This represents load against Memory Load and Store unit.
- **The number of bus activities.** Access to main memory, traffic for cache coherency, DMA operations, etc.

PMC driver

The `/proc/pmc` is a read-only file, and it provides information on the number of events measured by hardware performance monitoring counters. We developed the driver for the purpose of this project. Upon read, it prints the information after the last read, then resets the counters. For each process, it lists PID (process id), command name (the one seen in the ps command, for example), CPU ID (on which it is located), processor package ID, and performance event counters. For each logical processor, it also lists the accumulated event counts.

MASA calculates the following loads, reading the performance event counter information from the PMC driver.

Process Load

For a process, *process load* is calculated against its load metrics, based on the performance monitoring counters. Thus process load is not a linear value, but multi-dimensional one. Those bare values read from the performance monitoring counters maintained in the per-process data structure in the OS, and the PMC driver reads and reset those values.

CPU Load

CPU load represents the accumulated load against that logical processor, calculated by counts against the load metrics. Like a process, it is not linear value, but multi-dimensional one. Since there are many activities other than the ones for processes, such as interrupt, exception handling (especially, page fault), we don't calculate CPU load as the sum of process loads on it. Instead, we keep the performance monitoring counters running, and we update CPU load and reset the counters at context switch time. Those bare values read from the performance monitoring counters are maintained in the per-CPU data structure in the OS, and the PMC driver reads and reset those values.

4.1. Load Evaluation and Load Calibration

We need to evaluate the load to estimate the impact, especially when we migrate a process from a processor package to another, for the purpose of load balancing. We need to simplify the calculation and make it effective. There are two options for that:

Option 1 – Use a linear function converting a multi-dimensional value to a linear one. The issue with this evaluation is that we lose information when converting a multi-dimensional value to a linear one, especially which execution resources are used most for that process. It is possible that two different workloads can have a similar range of load under this evaluation. And they can have different impacts on the processor package because they utilize different set of execution resources

Option 2 – Check one load metric at a time. This is more effective, but we need to determine the order to check.

Check Each Load Metric and Load Calibration

The question with this option is the order in which we check imbalance at load balancing time. To set priority on the load metrics, we use what we call "load calibration." Load calibration determines the ratio of performance impacts caused by sharing a particular execution resource(s) for each load.

Example:

Run a typical floating-point intensive program on a logical processor in a processor package. Run the same program on the other logical processor, and measure the performance degradation (time) (ΔT_0) and the increased load (ΔL_0) associated with the floating-point load metric. The ratio $\Delta T_0/\Delta L_0$ basically provides relative impact of using floating-point execution resources.

Run a typical L2 cache intensive program on a logical processor in a processor package. Run the same program on the other logical processor, and measure the performance degradation (time) (ΔT_1) and increased number of events (ΔL_1) associated with the cache load metric.

The ratio $\Delta T_0/\Delta L_0 : \Delta T_1/\Delta L_1$ for example, indicates relative impacts of utilizing the executions resources associated with those load metrics. The load calibration technique can handle the variations of micro-architectures for a particular architecture, because we "measure" the actual impacts by running typical programs on the system. The micro-architectural scheduling assist determines the priority of load metrics at its initialization time.

Assuming we use the two programs FP (i.e. 177.mesa of SPEC-CPU2000) and L2C (197.parser of SPEC-CPU2000) for load calibration and 5-second samples reflect the load, we can use the data in Table 3.

First we need to get the performance of the Base cases (solo-run), and then run the test cases binding two instances of FP or L2C to a processor package (coupled-run) with the other processor package idle. For each run, we measure time (in sec.) and the event counts of FP uops and L2 cache misses.

From the data in Table 6, we can calculate $\Delta T_0/\Delta L_0 : \Delta T_1/\Delta L_1 : = 1.03$. This means an L2C cache miss event has slightly higher impacts. However, the above calculation is an example only, and we need to use pure benchmarks for this purpose.

If we use this data and look at Table 4, we realize that the processor package 1 has significantly high load (7.369E+07) in the (INT, INT) (L2C, L2C) test case.

4.2. Implementation

Algorithm for Resolving Load Balance

The basic method for resolving load imbalance between two packages (with the lowest and highest load) is:

1. Detect interference with execution resource(s) in a processor package.

   We use an *execution resource limit* measured at load calibration time. It is the value measured at solo run. An execution resource limit is the maximum number of the hardware events (per second) associated with a particular execution resource. For example, the execution resource limit for L2C is 46925899/244 = 192319 from Table 6. Note that it is an example to show how to calculate, and it may vary.

2. If found, find the processor packages with the lowest and highest load.

3. Look for a process to migrate, or two processes to swap in those processor packages, to equalize the loads as much as possible.

We employed swapping processes among processor packages rather than simple process migration, because of more choices when equalizing the loads. Since the process swapping mechanism can handle the process migration mechanism as well, we use process swapping hereafter.

Load Metric	Time (solo)	Time (coupled)	Load (solo)	Load (coupled,1/2)	Load (coupled, 2/2)	Load (coupled,total)	delta T	delta L
FP	134	252	106985316	58783608	60970800	119754408	118	12769092
L2C	244	445	46925899	31677006	36395793	68072799	201	21146900

Table: 6 Data for Load Calibration

In general we need to project the future load of the processors when migrating a process from a processor to another at load balancing time. Assuming that a process continues on the current workload, we can estimate the impact on the new processor that the processes are swapped. It's possible, however, that we can choose a wrong process at process swapping time, causing fluctuations.

We avoid unnecessary fluctuations by checking if the projected load after swapping does not exceed the execution resource limit for every execution resource.

MASA Program

MASA is a user-level program, and it requires the performance monitoring counters (PMC) driver (described below) that provides information of hardware monitoring counters. The following is a summary of the steps taken by the program:

1. Read the device file `/proc/pmc` implemented by the PMC driver, to get information of the performance monitoring counters for the last period.

2. Calculate process load for the existing processes, and calculate CPU load, and processor package load accordingly. Load of a processor package is given simply by adding the CPU load of the two logical processors in that package.

3. If we detect imbalance between the CPU loads in a processor package, for example, the case where one CPU has more than one outstanding process, but the other has none. This kind of situation can happen as new processes are spawned.

   We calculate (sum of package loads)/(4 × N) where N is the number of the CPUs in the system (4 is an ad hoc number obtained by experiments). If a process has load more than this value, then we increment the counter for the CPU.

   If any imbalance is detected, migrate one of the outstanding processes to the other CPU. Sleep for a specified time to refresh the performance counters, and go back to Step 1. Otherwise,

4. Begin with the top priority load metric:

   a. Check interference with the load, by checking if the load exceeds the execution resource limit. If not detected, then check the next load metric in the priority order. Go back to Step 4. Otherwise (if detected),

   b. Find the processor packages with the highest (`max_load`) and lowest CPU (`min_load`) load in the system with respect to the current load metric.

   c. If the imbalance is less than 25%, then jump to Step b (25% is an ad hoc value obtained by experiments). The imbalance is calculated by:
   `(max_load - min_load) / 2`

   d. If load imbalance is found, swap the processes that equalize the loads of the two processor packages with `max_load` and `min_load`, as much as possible by setting processor affinity. It uses *sched_setaffinity*() by providing the PID (process id) and the CPU mask (on which CPU the program must run) for them.

   If load imbalance is not detected or no processes to swap are found for this load metric, then go back to Step 4 for the next load metric. If all the load metrics are check, sleep for a time period specified (5 sec. by default), and go back to Step 1.

Note that migration can be suppressed by a command option to the program, and the program only logs such an event.

4.3. Performance

We made a prototype of MASA, and reached the same performance as the set up manually bound using full knowledge of the workload (using processor affinity statically), reducing performance variations. At this point, we are using information only on floating-point execution resources and L2 cache load and store misses.

SPEC-CPU2000

To verify the MASA mechanism works for various workloads, we simultaneously executed four shells of the following shells, each of which executes the programs from SPEC-CPU2000 sequentially:

- Shell_0 – 164.gzip, 176.gcc, 186.crafty, 252.eon, 254.gap, and 256.bzip2.

- Shell_1 – 175.vpr, 181.mcf, 197.parser, 253.perlbmk, 255.vortex, and 300.twolf.
- Shell_2 – 168.wupwise, 172.mgrid, 177.mesa, 179.art, 187.facerec, 189.lucas, and 200.sixtrack.
- Shell_FP – repeats 177.mesa 7 times to have the same duration as the other shells.
- Shell_L2C – repeats 197.parser 5 times to have the same duration as the other shells.

Shell_0 and Shell_1 are based on CINT2000 (for measuring and comparing compute-intensive integer performance), and the benchmark programs (12 of them) are split into these two shells one by one. Some of require large data areas.

Shell_2 is based on CFP2000 (for measuring and comparing compute-intensive floating-point performance), and the benchmark programs (7 of them) are picked alternatively i.e. one every two in the list. In reality, such floating-point intensive programs tend to use large data as well, and thus have higher L2C misses.

Note that obviously the SPEC-CPU2000 benchmarks are not intended to run this way, and thus our performance is not relevant to actual SPEC-CPU2000 performance. We measured the results from two test cases where the four shells are run in parallel on a system has dual Intel® Xeon™:

Test Case1 – Shell_0, Shell_1, Shell_2, and Shell_L2C

Test Case 2 – Shell_0, Shell_1, Shell_2, and Shell_FP

Shell	Benchmark	w/o MASA (Shell_L2C)	w MASA (Shell_L2C)
Shell_0	164.gzip	286	298
	176.gcc	561	598
	186.crafty	225	235
	252.eon	313	338
	254.gap	256	271
	256.bzip2	526	487
Shell_1	175.vpr	529	525
	181.mcf	607	613
	197.parser	630	515
	253.perlbmk	422	453
	255.vortex	479	458
	300.twolf	629	507
Shell_2	168.wupwise	337	331
	172.mgrid	708	639
	177.mesa	266	270
	197.art	842	781
	187.facerec	320	301
	189.lucas	314	299
	200.sixtrack	314	256
Shell_L2C	197.parser	519	498
	197.parser	597	587
	197.parser	466	505
	197.parser	557	563
	197.parser	336	389
Total		11036	10717

Shell	Benchmark	w/o MASA (Shell_FP)	w MASA (Shell_FP)
Shell_0	164.gzip	279	272
	176.gcc	475	408
	186.crafty	215	218
	252.eon	330	336
	254.gap	205	236
	256.bzip2	417	433
Shell_1	175.vpr	436	484
	181.mcf	474	448
	197.parser	481	536
	253.perlbmk	440	385
	255.vortex	473	290
	300.twolf	558	470
Shell_2	168.wupwise	297	300
	172.mgrid	621	513
	177.mesa	234	250
	197.art	585	594
	187.facerec	292	278
	189.lucas	279	196
	200.sixtrack	347	247
Shell_FP	177.mesa	244	246
	177.mesa	335	294
	177.mesa	302	284
	177.mesa	227	237
	177.mesa	251	263
	177.mesa	284	280
	177.mesa	244	245
Total		9322	8743

Table 7: Time (sec.) to complete SPEC-CPU2000 benchmarks w/o and w/ MASA

Table 7 shows the results without and with MASA enabled for the test cases, and Table 8 summarizes the results. The results are the average of three runs of each of the test cases (w/o and w/ MASA), and the numbers are the time to complete the benchmarks (in second). The time period for sampling used MASA is 5 seconds. Table 8 also shows the number of process migration during the test case, and they are applicable only to the cases with MASA. Test Case 1, for example, caused process swapping 51 times due to imbalance with respect to L2 misses, and 13 times due to imbalance with respect to FP uops. Since we placed higher priority on imbalance with L2 cache misses based on load calibration, we tend to see higher swapping rates for L2 misses.

At this point, we don't see visible overhead with running MASA mainly because it checks relatively infrequently (once per 5 sec. or, at most once per 1 sec.). Since our purpose is to detect and resolve severe load imbalance caused by execution resource contentions, we don't believe we need more frequent monitoring.

Test Case	Total (w/o MASA)	Total (w/ MASA)	Gain (%)	# of Swapping
Test Case 1	11036	10717	2.9	51 (L2 miss) 13 (FP uops)
Test Case 2	9322	8743	6.2	54 (L2 miss) 20 (FP uops)

Table 8: Summary of Table 7

5. Conclusions and Future Work

In a single processor package, HT enables the execution resources in a processor package to be used more efficiently. Excess resource bandwidth from one logical processor can be used to make progress on the other and increase overall utilization of the processor execution resources. In the multiprocessor (or multi-processor-package) environment, the OS can utilize the CPU execution resources among the processor packages with optimal placement of the processes, if it can monitor and balance the load of the execution resources utilized at runtime.

MASA uses the information from the hardware monitoring counters to evaluate the load of process packages and processes. We implemented a prototype of MASA, and the data shows that it improves performance by 6% for the workloads from SPEC-CPU2000, especially for floating-point intensive case. We still need to evaluate the effectiveness of MASA by running various applications, and to improve the algorithm, exploiting more load metrics.

I/O interrupt handling can be handled in the same fashion. At this point, the mechanism of I/O interrupt routing is primitive (typically done by the chipset and interrupt controller) and we believe that it can be improved by using the micro-architectural scheduling assist when deciding to which processor/package particular interrupts are routed at runtime.

Acknowledgement

We would like to thank the anonymous reviewers for their useful comments. We thank Sunil Saxena and Asit Mallick for their support.

References

[1] Guy G.F. Lemieux, "Hardware Performance Monitoring in Multiprocessors", Department of Electrical and Computer Engineering, University of Toronto,1996.

[2] Hinton, G; Sager, D; Upton, M; Boggs, D; Carmean, D; Kyker, A.; Roussel, P. "The Microarchitecture of the Pentum® 4 Processor", *Intel Technology Journal,* 2002. http://developer.intel.com/technology/hyperthread

[3] Intel® Vtune Performance Analyzer, http://www.intel.com/software/products/vtune/vtune60/

[4] W. L. Lynch, B. K. Bray, and M. J. Flynn, "The Effect of Page Allocation on Caches", Proc. of International Symposium on Microarchitecture, pp. 222-225, December, 1992

[5] Marr, D.; Binns, F.; Hill, D.; Hinton, G.; Koufaty, D.; Miller, J.; Upton, M. "Hyper-Threading Technology Architecture and Microarchitecture" *Intel Technology Journal,* 2002.

[6] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor", Proc. of International Conference on Measurement and Modeling of Computer Systems, June, 2002

[7] A. Snavely and D. M. Tullsen, "Symbiotic Jobscheduling a Simultaneous Multithreading Processor", Proc. of 9th International Conference on Architectural Support for Programming Language and Operating Systems, November 2000.

[8] M. S. Squillante and E. D. Lazowska, "Using Processor-Cache Affinity Information in Shared-

Memory Multiprocessor Scheduling", IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 2, February 1993, pp. 131 – 144.

[9] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", In ISCA96, pp. 191-202, May 1996

[10] D. Tullsen, S. Eggers and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism. In 22nd Annual International Symposium on Computer Architecture, pp. 392-403, June, 1995.

[11] B. Weissman, "Performance Counters and State Sharing Annotations: A Unified Approach to Thread Locality", Proc. of the 8th Int. Conf. Architectural Support for Programming Langurage and Operating Systems, pp. 127 – 138, 1998.

[12] SPEC CPU2000, http://www.spec.org/osg/cpu2000/

APPEDIX – Changes for HT in Linux

This appendix describes the changes for supporting HT in Linux 2.4.19 (www.kernel.org) kernel. Those changes or requirements should be applicable to an OS in general when supporting HT, although some of them might need to re-implemented for the target OS.

- fs/binfmt_elf.c (line 141 – line 155): Linux sets a constant value to the initial user stack pointer for every process. We adjust the pointer using the PID to minimize L1 cache eviction (see 1.2).
- kernel/sched.c (line 265 – line 279): Prioritize an idle package (both logical CPUs are idle) over an idle CPU, because the other logical CPU may not be idle, but busy. The variable smp_num_siblings maintains the number of the logical CPUs in a package. It is 2 for the Intel® Xeon™ processor family at this point.
- arch/i386/kernel/setup.c (line 2404 – line 2452): Detection of HT, and setup of mapping between the logical CPUs and the processor package, or the array `cpu_sibling_map[]`, which contains the other logical CPU number given the current CPU number as the index.
- arch/i386/kernel/acpitable.c (entire file): ACPI (Advanced Configuration & Power Interface, see http://www.acpi.info) table parsing. When HT is enabled, the number of the logical CPUs is reported in the ACPI table, not the MPS (Multi-Processor Specification, see http://www.intel.com/design/intarch/MANUALS/242016.htm, for example) table. The OS needs to look at the ACPI table for HT.
- arch/i386/kernel/semaphore.c (line 270, line 283): PAUSE instruction "rep; nop." This change is useful for generic IA-32 Intel® Architecture SMP systems as well.
- arch/i386/kernel/mtrr.c: The code used when if the target CPU is MTRR_IF_INTEL. Since the MTRRs (Memory Type Range Register) resources are shared in a processor package, and update to MTRRs must be atomic. In Linux, it was possible that two logical CPUs in processor package update MTRRs simultaneously and changed those registers inconsistently.
- arch/i386/kernel/microcode.c (line 76, line 252 – line 299): The current Linux updates the micro code simultaneously on every CPU sending Inter-Processor Interrupt (IPI). Since the micro code resource is shared by the logical CPUs in a processor package, the update should be done one time. To ensure that the update happens only one time for each processor package rather than for each logical CPU, the spin lock

  `microcode_update_lock` is used to make the update atomic. Otherwise we see a race condition.
- include/asm-i386/spinlock.h (line 62): No change required. Original code already had had this code (`PAUSE` instruction).
- include/asm-i386/system.h (line 318): No change required. Original code already had had this code (`hlt` instruction for the idle loop).

WIESS '02
3:30 p.m.–4:30 p.m.

Session Chair:
Rob Gingell, *Sun Microsystems, Inc.*

An Examination of the Transition of the Arjuna Distributed Transaction Processing Software from Research to Products

M. C. Little[1] and S. K. Shrivastava[2]
[1]*HP-Arjuna Laboratories, Newcastle-Upon-Tyne, UK*
[2]*Department of Computing Science, Newcastle University, Newcastle-Upon-Tyne, UK*

Abstract

The *Arjuna* transaction system began life in the mid 1980s as an academic project to examine the use of object-oriented techniques in the development of fault-tolerant systems; over 15 years later it is now a Hewlett-Packard product in its own right and is also embedded in several other offerings from HP. In addition, many of the original developers of Arjuna have accompanied the system on its journey and had first hand experience in taking this academic research vehicle into a commercial environment. At times the transition has been neither easy nor smooth but it has been interesting from many different perspectives. In this paper we shall attempt to give an overview of how this occurred and illustrate some of the lessons we have learned over the years.

1. Introduction

The *Arjuna* transaction system began life in the mid 1980s as an academic research project to examine the use of object-oriented techniques in the development of fault-tolerant distributed systems; 15 years later it is a product in its own right and is also embedded in several other product offerings. In this paper we shall examine how this transition was achieved and the lessons learnd along the way.

Arjuna is an object-oriented programming system that provides a set of tools for the construction of fault-tolerant distributed applications. *Arjuna* supports the computational model of *nested atomic actions* (nested transactions) controlling operations on persistent objects. *Arjuna* objects can be replicated on distinct nodes for obtaining high availability. The *Arjuna* research effort began in late 1985 at the University of Newcastle. A version of the system written in C++ to run on networked Unix systems was operational in the early nineties and maintained and made available freely on the Web for research, development and teaching purposes. The arrival of the Web and industrial acceptance of CORBA and Java technologies for distributed object computing during this period encouraged us to productise *Arjuna*. In late 1998 we set up a company, Arjuna Solutions Ltd., with two products derived from *Arjuna*: OTSArjuna, a C++ version of the CORBA Object Transaction Service (OTS) and JTSArjuna, the OTS counterpart in Java. Through a series of company acquisitions, these later became part of HP's middleware product lines. Within HP, the original Arjuna software continues to be of use in creating customised transactional services for new application areas, such as Web Services and mobile computing. In this paper we examine the reasons for the longevity of the Arjuna software.

The paper is structured as follows. In the next section we present an overview of the *Arjuna* system that was implemented in C++. Sufficient details of the system are presented here to enable the readers to follow the subsequent discussions concerning middleware. The material of this section is taken from our published papers on Arjuna [1,2,3,4]. Section three describes how the system was adapted for use as a transaction service for CORBA and Java middleware; here we also compare and contrast the functionality of the original Arjuna system with that of the modern component based middleware. Section four concludes the paper with some lessons we have learnt from our experiences.

2. An Overview of Arjuna

2.1. Design and Implementation Goals

The design and implementation goal of *Arjuna* was to provide a state of the art programming system for constructing fault-tolerant distributed applications. In meeting this goal, three system properties were considered highly important:

(i) *Modularity:* The system should be easy to install and run on a variety of hardware and software configurations. In particular, it should be possible to replace a component of *Arjuna* by an equivalent component already present in the underlying system. As we shall show in the rest of this paper, the modularisation aspect is an important factor in

Arjuna's longevity. At the time, this was accomplished through the use of relatively new object-oriented techniques of interface-implementation separation: each module, which defines a specific well-defined set of functionality, is interacted with through an interface without exposing the underlying implementation details.

(ii) *Integration of mechanisms:* A fault-tolerant distributed system requires a variety of system functions for naming, locating and invoking operations upon local and remote objects, and for concurrency control, error detection and recovery from failures, etc. These mechanisms must be provided in an integrated manner such that their use is easy and natural.

(iii) *Extensibility:* These mechanisms should also be *flexible,* permitting application specific enhancements, such as type-specific concurrency and recovery control, to be easily produced from the existing default ones.

The first goal was met by dividing the overall system functionality into a number of modules that interact with each other through well defined *narrow* interfaces. This facilitated the task of implementing the architecture on a variety of systems with differing support for distributed computing. For example, it was relatively easy to replace the default RPC module of *Arjuna* by Sun RPC and to provide persistence implementations ranging from flat file to non-volatile RAM based. The remaining two goals were met primarily through the provision of a C++ class library for incorporating the properties of fault-tolerance and distribution. Finally, and purely for pragmatic reasons, we decided that it was important to develop *Arjuna* using commonly available tools and hardware: being an academic institution, we had very few funds on which to call for projects.

2.2. Objects and actions

Arjuna supports a computation model in which applications manipulate persistent objects under the control of atomic transactions. Distributed execution is achieved by invoking operations on remote objects using remote procedure calls (RPCs). All operation invocations may be controlled by the use of transactions, which have the well known ACID properties (Atomicity, Consistency, Isolation, Durability). Transactions can be nested; nesting provides fault-isolation: a nested action can abort without causing the abortion of the enclosing action. A (two-phase) commit protocol is used during the termination of an outermost atomic action (*top-level action*) to ensure that either all the objects updated within the action have their new states recorded on stable storage (committed), or, if the transaction aborts, no updates get recorded. It is assumed that, in the absence of failures and concurrency, the invocation of an operation produces consistent state changes to the object. Transactions then ensure that only consistent state changes to objects take place despite concurrent access and any failures.

2.3. System Architecture

With reference to Fig 1, we shall now identify the main modules of *Arjuna* and the services they provide for supporting transactional persistent objects.

1) *Atomic Action.* Provides atomic action support to applications in the form of operations for starting, committing and aborting transactions. It provides a high level API called the *Arjuna Integrated Transactions* (AIT).

2) *RPC.* Provides facilities to clients for connecting/disconnecting to object servers and invoking operations on objects. The initial implementation of this was developed within the Arjuna research group [5] and used novel C++ stub-generation techniques to enhance distribution transparency [6].

3) *Object Store.* Provides a stable storage repository for persistent objects; these objects are assigned unique identifiers (Uids) for naming them.

4) *Naming and Binding.* Provides a mapping from user-given names of objects to Uids and a mapping from Uids to location information such as the identity of the host where the server for the object can be made available.

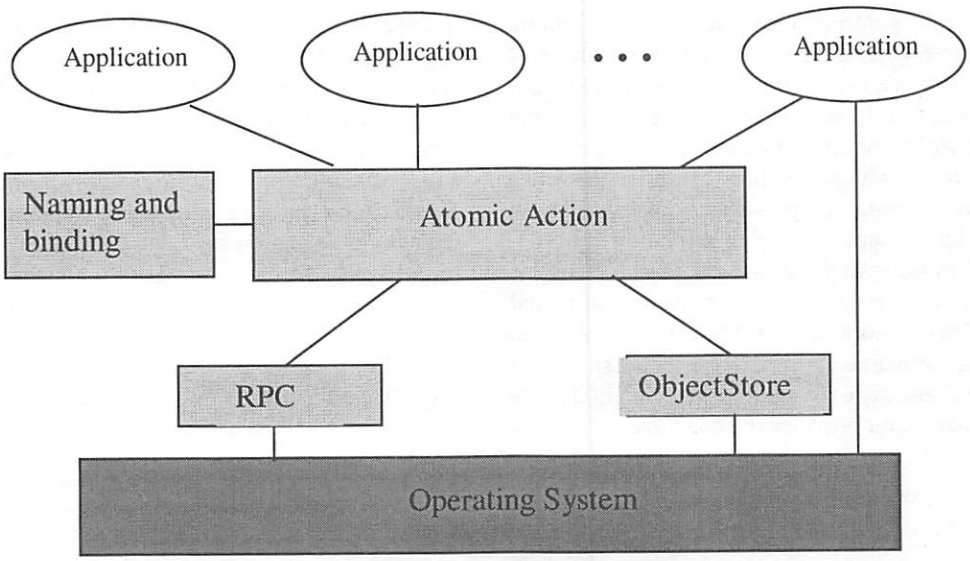


Fig. 1: Components of *Arjuna*.

Every node in the system provides the RPC and Atomic Action modules. Any node capable of providing stable object storage in addition contains an Object Store module. Nodes without stable storage may access these services via their local RPC module. The Naming and Binding module is not necessary on every node since its services can also be utilised through the services provided by the RPC module.

Although atomic transactions guarantee consistency in the presence of failures, they do not provide a means of guaranteeing forward progress: the failure of a machine can lead to the abortion of the transactions using that machine Therefore, as part of the Object Store module, *Arjuna* provides a high-availability option that allows persistent object states to be replicated on an arbitrary number of machines, thus improving the probability that machine failures will not cause a transaction to abort.

All of these modules are accessed through interfaces that allow their implementations to be replaced at runtime. For example, there are multiple interfaces between the Atomic Action and RPC modules offering different, pluggable functionality. When making invocations on remote objects which occur within transactions, it is necessary to propagate information about the transaction (the *context*) to the remote Atomic Action module. Therefore, there are interfaces for allowing the RPC component to obtain and serialise the context on outward calls and to de-serialise the context and recreate the transactions it refers to on incoming calls. The Object Store module has multiple different implementations, each tailored to a specific mode of use (e.g., non-volatile RAM, flat file space or database) and these can be selected on a per object basis.

2.4. Coordinating Recovery, Persistence and Concurrency Control

The atomic action module is the most important part of *Arjuna*. Its design is based on the principle that as objects are assumed to be encapsulated entities, then they must be responsible for implementing the properties required by atomic actions themselves (with appropriate system support). This enables differing objects to have differing recovery and concurrency control strategies. Given this proviso, any atomic action implementation need only control the invocation of the operations providing these properties at the appropriate time and need not know how the properties themselves are actually implemented.

The principal classes that make up the class hierarchy of Arjuna Atomic Action module are depicted in Fig. 2. This is an important hierarchy because, as we shall see, certain classes within it (e.g., `AbstractRecord`) have been critical in the longevity of Arjuna. In addition, the relative ease with which complex applications can be developed using this hierarchy has also helped in Arjuna's success.

To make use of atomic actions in an application, instances of the class `AtomicAction` must be declared by the programmer; the operations this class provides (`Begin`, `Abort`, `End`) can then be used to structure atomic actions (including nested actions).

The only objects controlled by the resulting atomic actions are those objects which are either instances of Arjuna classes or are user-defined classes derived from `LockManager` and hence are members of the hierarchy shown. Most Arjuna classes are derived from `StateManager`, which provides primitive facilities necessary for managing persistent objects. These facilities include support for the activation and de-activation of objects, and state-based object recovery. Thus, instances of `StateManager` are the principal users of the Object Store module and isolate users from its underlying implementations (e.g., database or file based). `LockManager` uses the facilities of `StateManager` and provides the concurrency control required for implementing isolation. The implementation of atomic action facilities for recovery, persistence management and concurrency control is supported by a collection of object classes derived from `AbstractRecord` which is in turn derived from `StateManager`. For example, instances of `LockRecord` and `RecoveryRecord` record recovery information for Lock and user-defined objects respectively. `AtomicAction` manages instances of these classes (using an instance of the class `RecordList` which corresponds to the intentions list used in traditional transaction monitors) and is responsible for performing aborts and commits.

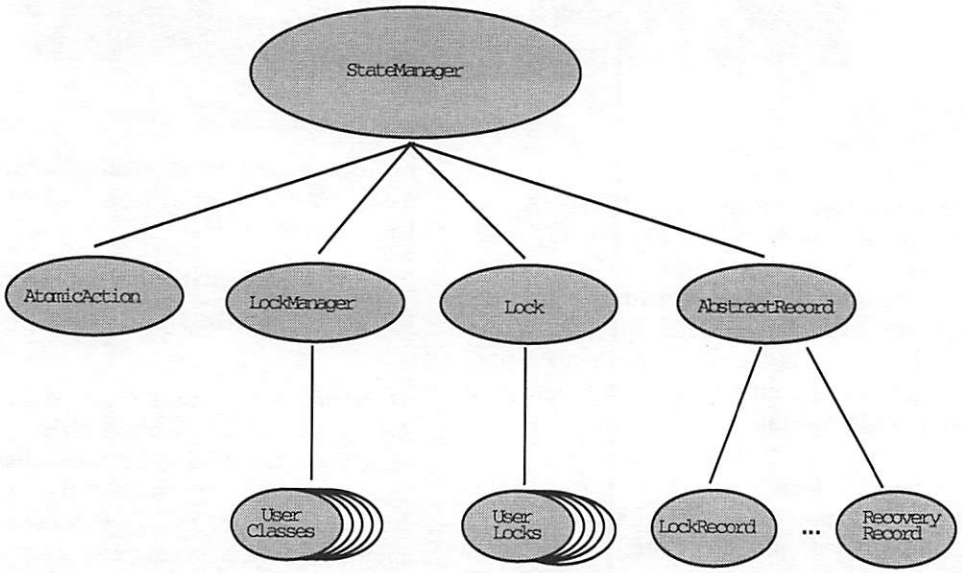


Fig. 2: The Arjuna Class Hierarchy.

Consider a simple example. Assume that `O` is a user-defined persistent object. An application containing a transaction `A` accesses this object by invoking operation `op1` of `O` which involves state changes to `O`. The serialisability property requires that a write lock must be acquired on `O` before it is modified; thus the body of `op1` should contain a call to the appropriate operation of the concurrency controller (see Fig. 3):

```
{
 // body of op1
     if setlock (new Lock(WRITE)
=== GRANTED)
     {
       // actual state change
operations follow
        ...
     }
}
```

Fig. 3: The use of Locks in implementing operations.

The operation `setlock`, provided by `LockManager`, performs the following functions in this case:

(i) check write lock compatibility with the currently held locks, and if allowed,

(ii) use `StateManager` operations for creating a `RecoveryRecord` instance for `O` (this is a `WRITE` lock so the state of the object must be retained before modification) and insert it into the `RecordList` of `A`;

(iii) create and insert a `LockRecord` instance in the `RecordList` of `A`.

Suppose that action `A` is aborted sometime after the lock has been acquired. The `abort` operation of `AtomicAction` will process the `RecordList` instance associated with `A` by invoking the `abort` operation on the various records. The implementation of this operation by `LockRecord` will release the `WRITE` lock while that of `RecoveryRecord` will restore the prior state of `O`.

The `AbstractRecord` based approach of managing object properties has proved to be extremely useful. Several uses are summarised here. `RecoveryRecord` supports state-based recovery, since its abort operation is responsible for restoring the prior state of the object. However, its recovery capability can be altered by refining the abort operation to take some alternative course of action, such as executing a compensating function. `LockRecord` is a good example of how recoverable locking is supported for a Lock object: the `abort` operation of `LockRecord` does not perform state restoration, but executes a `release_lock` operation. Similarly, no special mechanism is required for aborting an action that has accessed remote objects. In this case, instances of `RpcCallRecord` are inserted into the `RecordList` as RPCs are made to the objects. Abortion of an action then involves invoking the `abort` operation of these instances which in turn send an "abort" RPC to the servers.

In keeping with the tradition of a university research group, the system as described above was developed and used by several graduate students (including the first author) as a part of their doctoral research work, which included building a distributed database engine for book citations. *Arjuna* was also used in a number of industrial research projects [2], for example use in the telecoms arena for managing bandwidth reservation and connection setup. A particularly demanding application has been the electronic student registration system in use since 1994 by Newcastle University [8]. The registration system has a very high availability and consistency requirement; admissions tutors and secretaries *must* be able to access and create student records (particularly at the start of a new academic year). In addition, the University required any solution to be software based and to run on existing hardware and operating systems, including Unix, Microsoft Windows and MacOS. At that time, no other software based solution existed that could fulfil all of those requirements. Arjuna provided the right set of mechanisms: transactions for consistency and replication for availability. In this particular application, the student record database was triplicated. During the 8 years that the system has been in use, there have been several network and machine failures; with one exception (see below), Arjuna has coped with them all, leaving users unaware that anything untoward has occurred.

The student registration system is composed of two sub-systems: the 'Arjuna sub-system' that runs on a cluster of Unix workstations and is responsible for storing and manipulating student data using transactions, and the 'front-end' sub-system, the collection of PCs and Macs each running a menu driven graphical user interface that users employ to access student data through the Arjuna sub-system The Arjuna-subsystem was engineered to run in a non-partitionable environment by ensuring that the entire cluster of machines was on a single, lightly loaded LAN segment; this decision was made to simplify the task of consistency management of replicated data (as can be appreciated, the problem of consistency management is quite hard in a partitionable environment). The current Arjuna sub-system configuration consists of ten Unix workstations, of which three act as a triplicated database (object store) Within the Arjuna sub-system, great care was taken to ensure that safely (pessimistically) chosen timeouts together with network level 'ping' mechanisms do act as an accurate indication of node failures. During the first year of deployment, we did experience initial consistency problems when one of the replica was inaccurately diagnosed as failed; this led to further testing and adjustment of failure detection timeouts.

Success in meeting the requirements of the registration system was one of the factors that led the Arjuna group to consider turning the system into a product in 1996. By then CORBA and Java middleware were attracting industry attention. The OMG architecture, CORBA [9] was well established, so it seemed natural to adapt *Arjuna* to meet the specification of the Object Transaction Service (OTS) and later to the Java Transaction Service (JTS) that is optional within the Java component middleware, J2EE [10]. In the next section we describe how this was achieved.

3. Arjuna and Middleware

3.1. Basic middleware concepts

We present a brief overview of middleware concepts, using CORBA as an example. Fig. 4 depicts the main elements of the CORBA middleware. It consists of an 'object bus', the object request broker (ORB) which allows client to interact with remote objects. A number of services are available for facilitating this task; these include naming, persistence, event notification and transactions. JAVA/RMI is a broadly similar Java language specific middleware.

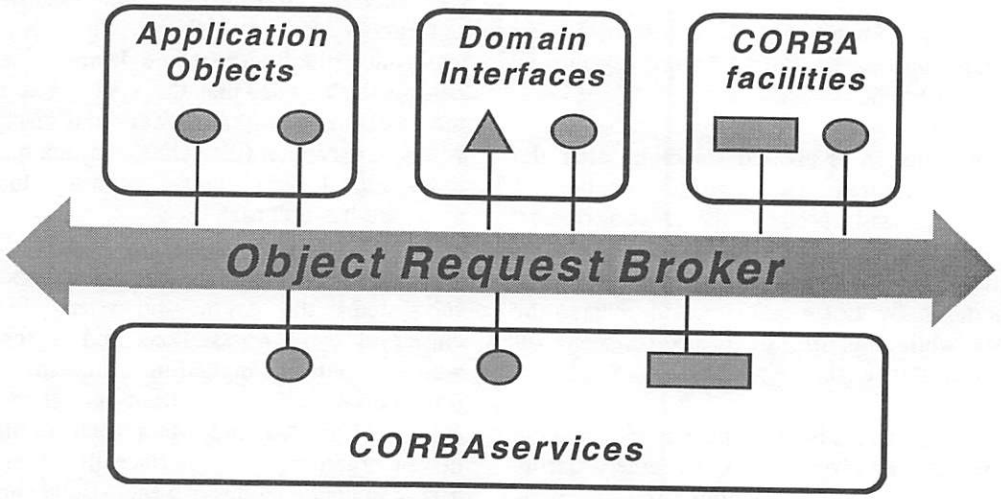


Fig. 4: CORBA middleware.

Although this middleware simplifies the construction of distributed applications by providing type checked remote invocations and standard ways of using commonly required services, there is still the problem that programmers have to worry about application logic as well as technically complex ways of using a *collection of services*. For example, transactions on distributed objects require concurrency control, persistence and the transaction services to be used in a particular manner. To address this difficulty, object-based middleware has been extended to component-based middleware. In simple terms, a component is an 'application object with the capability for using middleware services in a standard manner'. A component is hosted by a *container* (a server process), and normally the container uses the underlying middleware services on behalf of the application object. A component descriptor specifies, in a declarative manner, the services that are required by the component. Containers are provided by *application servers* that provide tools for deploying components onto containers using the information specified in the descriptors. A J2EE Enterprise Java Bean (EJB) is a good example of a component. In the rest of this section we will first examine how OTSArjuna and JTSArjuna were implemented. Secondly, bearing in mind that Arjuna is not just a transaction service, but a complete system for building transactional applications, we will compare and contrast its functionality with that of J2EE application servers.

3.2. Arjuna, the OTS and the JTS

In 1995 the industry standard for distributed transactions changed from being predominately based on X/Open XA [7], which is procedural-oriented, to the OTS (OMG Object Transaction Service) [9]. This was based on the experiences of all of the major transaction processing vendors, including IBM, HP and DEC. Around this time, the *Arjuna* group attracted R&D funding from industry led consortia and members of these consortia expressed interest in standards-related developments. In particular several of our industrial sponsors were interested in funding transactions research and development within a CORBA environment and the original Arjuna system, with its own RPC and Naming and Binding implementations, did not meet these requirements.

The OTS is a *protocol engine* that guarantees that transactional behaviour is obeyed but does not directly support all of the transactional properties. As such it requires other services that implement the required functionality, such as persistence and concurrency control. The application programmer is responsible for using these services to ensure that transactional objects have the necessary ACID properties.

In addition, the OTS does not provide any participant implementations (e.g., database resource managers). These must be provided by the application programmer or the OTS implementer. As such, a pure OTS implementation actually provides much less functionality than that available in *Arjuna*. The OTS does not define a complete toolkit for the construction of transactional applications: it has no equivalent of AIT.

Examining just the transaction engine component of *Arjuna* as provided by the Atomic Action module, it was clear that there was a good match with the OTS. Although the interfaces that are exposed by the OTS were different, it was relatively straightforward to provide these same abstractions on top of the equivalent *Arjuna* APIs. Most effort was directed at fully integrating *Arjuna* (now *OTSArjuna*) within the CORBA framework, e.g., how to do distributed invocations and ensure that the transaction context is passed as mandated by the specification. The interfaces we had defined between the Naming and Binding and RPC modules were sufficiently powerful that only minimal modifications had to be made. In addition, the interfaces allowed *OTSArjuna* to be ported to a variety of different ORBs rather than being tied to one specific implementation. This was an important aspect as other OTS implementations required users to own a specific ORB (usually the one from the same vendor), e.g., IBM WebSphere and BEA WebLogic. Being able to choose the vendor of different components in a distributed architecture was important to many prospective users.

The architecture of *OTSArjuna* is shown in Fig 5, where the dark grey boxes comprise *OTSArjuna* and everything else is either provided by the application programmer/user or other software components such as database drivers (Resource/SubtranAware, for example). The OTS protocol engine, State Management and Concurrency Control components are essentially exactly as they appeared in the original *Arjuna* Atomic Action module. AIT was provided to programmers via the *OTSArjuna* API. The RPC and Naming and Binding modules from the original architecture were replaced by whatever the underlying ORB implementation provided. The existing Object Store module was made available via suitable Resource/SubtranAware implementations.

Importantly, to maintain the original pluggable abstraction, any OTS-specific modifications that were made (such as API updates) occurred in self-contained modules accessed through well defined interfaces.

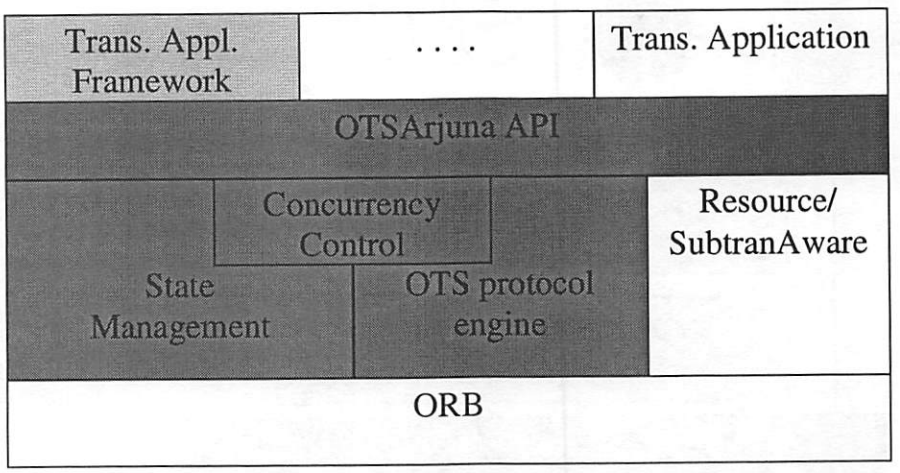


Fig. 5: OTSArjuna architecture.

Remote participants, XA compliant databases etc. were all transparently controlled by the core via `AbstractRecord` implementations: the transaction protocol engine could not tell that it was now running within a CORBA environment. However, there was one notable exception where the original *Arjuna* abstraction we had simply did not work: *failure recovery*.

To guarantee ACID properties in the event of failures, a recovery subsystem is required. This ensures that any transactions that were in progress are completed, either by being committed or rolled back. In order to do this, it may be necessary to recreate any resources that were participants within the transaction: the recovery system will recreate the distribution tree that was present prior to the failure. In order to achieve this, recovery must have intimate knowledge about the resources (e.g., do

they use a file system for persistence?) and the RPC mechanism (e.g., what is the machine the object resides on?) The *Arjuna* failure recovery implementation was closely tied to the original RPC mechanism. We were therefore unable to take this component or to reuse the interfaces it provided. As such, we were forced to re-implement recovery and in doing so we tied it to the OTS model for expediency.

By 1996 Java started attracting serious attention from industry and many existing OMG standards made their way into J2EE. Critical amongst them was the OTS. *OTSArjuna* was developed using C++, which made the task of converting to Java relatively straightforward. Arjuna had been developed with the very earliest versions of C++, which had no multiple inheritance and hence this helped facilitate the language transition. We had made extensive use of C++ templates and reference parameters, neither of which had their equivalents in Java at that time and this did require us to expend effort in re-coding. However, none of the language differences presented issues that were substantial enough to require redesigning large portions of the code.

JTSArjuna, became the worlds first 100% pure Java transaction system. This system and the group which helped develop it became part of Hewlett-Packard's middleware division through a series of acquisitions. After two years of intensive work on turning an academic system into a product (mainly increasing the team from 5 to 24 people, of which nearly 40% were dedicated to quality assurance, and putting the product through much more rigorous testing than it had ever been through before), the *Hewlett-Packard Transaction Service* was created and became an integral part of their middleware offerings. Interestingly, because of its long history of development and use, the number of issues (bugs) thrown up by quality assurance was relatively small for such a complex system.

3.3. Arjuna and J2EE

In many ways, the programming model and environment presented by Arjuna has many things in common with modern day application servers. In this section, we shall compare and contrast Arjuna with the J2EE application server architecture and try to have an objective examination of why differences exist.

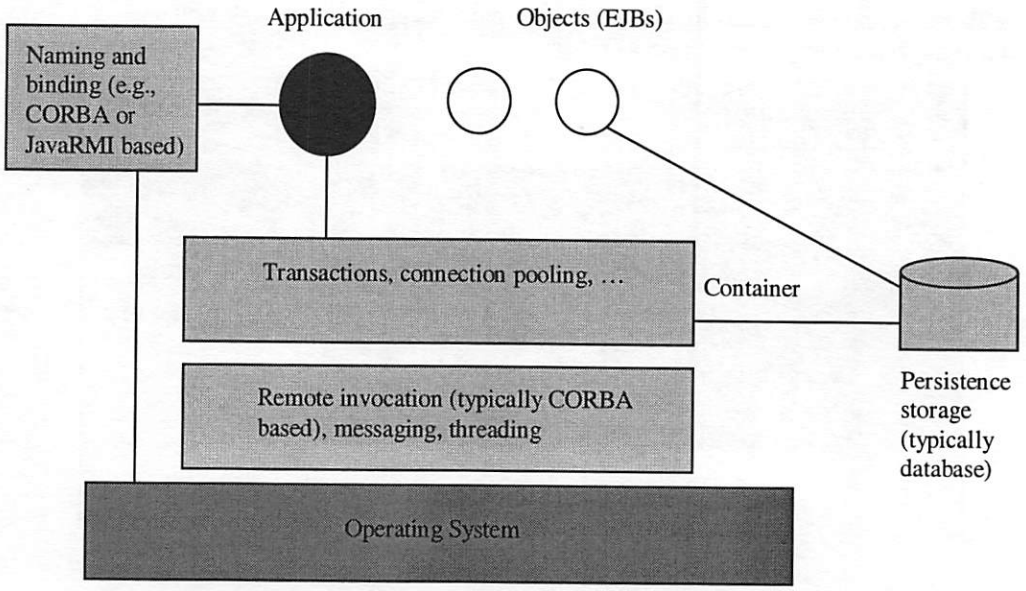


Fig. 6: Simplified application-server architecture.

A simplified J2EE application server architecture is shown in fig. 6. Although this depiction hides many details, it should be sufficient for a high-level comparative analysis. It has very similar components to *Arjuna*, but utilises industry standard technologies for the object invocation mechanism and naming and binding. Although the interfaces to the various components are different to those in *Arjuna*, the fact that these components exist in an identifiable (and typically replaceable) manner is testimony that the original *Arjuna* architecture and design goals were sound.

A feature that is missing in *Arjuna* is the declarative way of managing transactions that is provided by EJBs.

EJBs also provide explicit transaction management using a high level (compared to JTS) API, called Java Transaction API (JTA); however, it is not as convenient to use as AIT, and its use is not generally recommended [11]. Compared to *Arjuna*, EJB transactions come with several restrictions:

(i) *Participant restriction*: both the original *Arjuna* system and the OTS, allow arbitrary participant implementations. The AbstractRecord interface and the OTS equivalent do not mandate a specific implementation. This allows recovery, concurrency control etc. to be transparently enlisted with a transaction. However, the JTA interface restricts participants to being XA-aware. This effectively mandates that applications must use databases for persistence: other types of persistence are possible, but a lot of effort is necessary from the developers in order for the implementations to be driven by the JTA, since whatever persistence implementation is used, it must be XA compliant. Several users of *Arjuna* and its descendant products have found the fact that persistence implementations are configurable and not bound to a specific model (e.g., XA) extremely useful, especially in the area of ease of deployment.

(ii) *No nested transactions*: if an object's methods are required to use transactions then, in an environment which supports nested transactions, the implementer can use transactions without concern about how those methods will be used: if the invoker uses transactions, then the methods will be nested within them, otherwise they will be top-level. In addition, nesting provides a level of failure containment, since the failure of a nested transaction does not require the enclosing transaction to roll back. The JTA does not support nested transactions because XA does not. Once again, user feedback on nested transactions has shown that they are an extremely useful structuring tool, especially when developing large-scale systems which may have many engineers working on them.

(iii) *Poor concurrency control*: Unfortunately, no satisfactory way of using read and write locks is available in EJBs. Because the JTA mandates that all participants must be XA-aware, this ties the persistence model to using a database. Most databases implicitly couple persistency and concurrency together, such that, for example, when an object loads its state it obtains a lock on the entire database table, which is maintained for the duration of the transaction. All other object states held within the same table are also implicitly locked. In order to provide object-level concurrency control, the programmer is supposed to make use of the Java language `synchronized` construct, which obtains an exclusive lock on a method. However, this construct is not transaction-aware and as such cycles (where object A calls object B which calls object A) can result in deadlock and are illegal. Because AIT locks are transaction-aware, not only can an object use multiple-reader/single-writer policies, but cycles within a transaction are supported. This makes the construction of complex, distributed applications more straightforward as programmers need not worry about whether cycles may occur, which could require in-depth knowledge of objects implemented by others.

(iv) *No support for orphan detection and elimination*: Client crashes or network partitions can occasionally create orphan servers.. Orphans are undesirable as they consume resources, and need to be destroyed. The RPC mechanism used in *Arjuna* detected and eliminated orphans [5]. Experience with the use of application servers has indicated that orphans do occur in practice; unfortunately no automatic support for orphan detection and elimination is provided in application servers (or any other middleware system for that matter). Given our experience with the Student Registration system and phantom machine failures, this lack of orphan detection is worrying.

4. Further evolution

The increasing use of the Web for commercial activities has led to a paradigm shift from closely-coupled, synchronous systems to loosely coupled asynchronous systems. Several researchers (e.g. [12,13]) have argued that in the Internet/Web environment, a practical way of gluing applications is through loose coupling as provided by asynchronous messaging. The main reason behind this is that asynchronous communication de-couples producers of information from consumers; they do not need to be both ready for execution at the same time.

The one constant amongst traditional and new distributed environments is their requirement for transactions. Irrespective of whether applications are closely-coupled or loosely-coupled, failures happen that affect both the performance and consistency of applications run over them. Transactions can be used in all of these environments to ensure consistency and specifically within Hewlett-Packard, Arjuna transaction technology has been used.

4.1. ArjunaCore

As part of the Hewlett-Packard NetAction product suite, there was a requirement for both transactional messaging (based on the JMS specification) and Web

Services transactions implementations (based on the OASIS Business Transactions Protocol [14]). The existing transaction products covered the J2EE and CORBA arenas using a two-phase commit protocol. When comparing the functionality provided by *JTSArjuna* with that required by the JMS, for example, it was clear that there was much overlap. In fact, it was possible to categorise all of the transaction products with the following requirements:

- The use of a two-phase commit protocol.
- They carry transaction context information in a manner suitable to their environment, e.g., XML and SOAP for BTP, or IIOP for CORBA.
- Their transaction participant implementations are opaque to the two-phase transaction engine.

Careful examination showed that it appeared possible to use the same *core* protocol engine that had been within the original *Arjuna* system and was now within *JTSArjuna*, within HP's BTP (HP Web Services Transactions) and JMS (HP Messaging Service) implementations. The interfaces (e.g., `AbstractRecord`) isolate the coordinator from the specifics of participant implementations and how transactional distributed invocations occur. By providing different implementations of`AbstractRecords`, for example, it was possible to drive BTP Web Service participants or JMS participants through a two-phase commit protocol using the *exact* same transaction engine.

Therefore, the work that was performed to transform *Arjuna* into *JTSArjuna* was generalised. The first step was to create a fully-functional transaction engine that had *no* dependencies on CORBA (including failure recovery) or any distribution infrastructure. This created *ArjunaCore*, which is concerned solely with the use of local transactions, i.e., transactions that run on a single machine. If distributed transactions are required, *ArjunaCore* provides the necessary hooks to enable information about its transactions to be transmitted in a manner suitable for the environment in which it is running, e.g., CORBA IIOP or SOAP/XML. Systems that use *ArjunaCore* for their transaction requirements are then required to utilise these hooks in order to obtain the context information and then transmit it in a suitable manner.

The main obstacle to the design of *ArjunaCore* was the failure recovery sub-system. When designing *OTSArjuna* it had been closely tied to the CORBA OTS model. In order to determine transaction statuses, it was a requirement that *all* transactions were implemented by CORBA objects, whether or not they were used in a distributed manner. By re-examining the failure recovery architecture, a more modular approach was taken, that was essentially based on the original Arjuna goals: recovery is directed through an abstract interface (the RecoveryModule) that does not imply specific implementations. Each RecoveryModule is responsible for recovering specific types of resources (transactions, application objects etc.) without exposing implementation choices such as whether or not CORBA was used, to the recovery framework. Since *ArjunaCore* is responsible for only local transactions, its default RecoveryModule implementations are relatively simple. Other products in which *ArjunaCore* is embedded, e.g., BTP, provide suitable implementations to do distributed recovery where necessary.

With the benefit of hindsight, this abstraction should have been used from the start when creating *OTSArjuna*. The reason it was not was basically that, in an effort to productise the system we believed that the OTS was to be the final destination for *Arjuna*: CORBA was being widely adopted as the de facto middleware standard and J2EE helped to propagate that myth. As such, there did not seem to be any reason to believe that Arjuna would be used anywhere except within an OTS environment and hence no requirement for such flexibility in the recovery sub-system. Obviously this did not turn out to be the case and we paid the price in time and effort spent in re-implementation and backward compatibility support.

Despite the necessary redesign of the failure recovery subsystem, the remainder of *ArjunaCore* is the same as existed in the original *Arjuna* system. The interfaces have proven sufficient to allow the system to be the core transaction engine within a diverse range of products. The fact that it is no longer reliant on a specific distribution infrastructure makes it extremely small (less than 250 kilobytes compared to almost 1 megabytes for *JTSArjuna*) and embeddable: experiments have been performed to port ArjunaCore onto mobile devices (PDAs) something which other industrial transaction products would find extremely difficult to accomplish.

5. Concluding remarks

Thus, after nearly 15 years of design and evolution, Arjuna has evolved through its various OTS/JTS incarnations, to a stand-alone, non-distributed transaction engine. Of the original architecture shown in fig 1., only the Atomic Action and ObjectStore

modules remain. However, of these, the majority of the code that was written while in academia continues to exist within the *ArjunaCore* product.

Figure 7 attempts to illustrate the history of Arjuna as a timeline, showing the relevant events we have discussed previously from its start in 1986 to the present day.

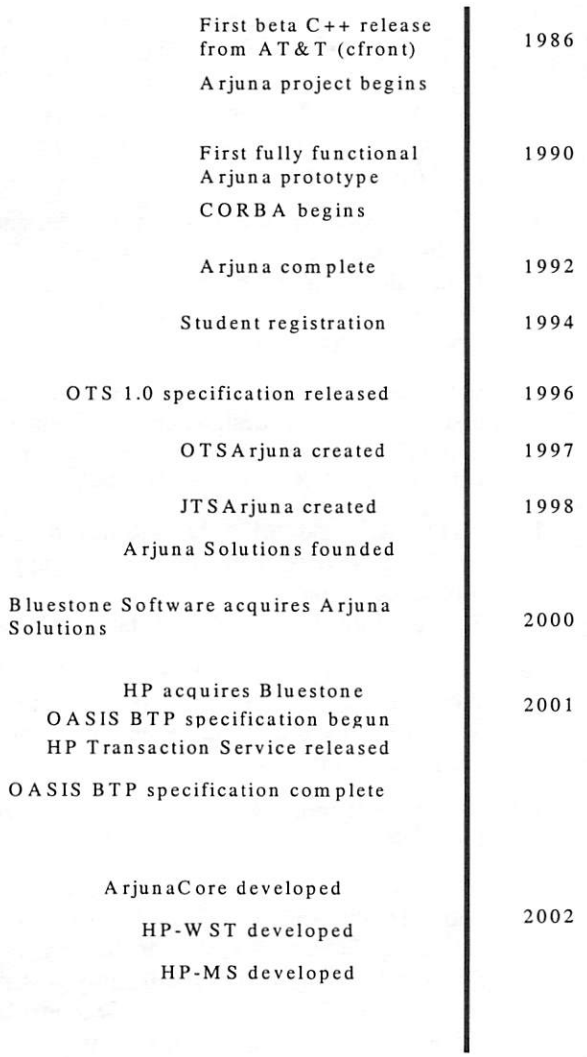


Fig 7. The evolution of Arjuna.

In achieving the transition of the Arjuna distributed transaction processing software from research to products, we have learned a number of lessons, some of which will be relevant to others involved in or embarking on a similar process. We shall attempt to enumerate them below:

- Modularity within the architecture helped us to restructure the uses to which we put Arjuna without requiring re-implementation of the entire system. As we have discussed, the core transaction engine available today remains relatively unchanged from its original C++ version.

- The use of object-oriented techniques helped to make the structuring of the architecture flexible and extensible. It also helped to make its use relatively intuitive for new developers. A crucial factor has been the structure of the atomic action module for coordinating concurrency, persistence and recovery for atomic actions using `AbstractRecords` (section 2.4), which meant that transaction coordinator need only control the invocation of the operations providing these properties at the appropriate time and need not know how the properties themselves are actually implemented.

- A commercial product requires a lot more emphasis on quality assurance (QA) and testing processes than a research system. At the time of writing, the number of QA tests for Arjuna number in the thousands, cover every aspect of the system and can take days to run to completion. Only with the evidence of these tests is it possible to convince people to invest time and money in purchasing the product.

- Within each component there are typically many places where configuration choices are made (e.g., the location of the object store, the maximum size of the transaction log before the system begins to prune it, etc.) When Arjuna started, many of these choices were hardwired in at compilation time. Over the years (and particularly when it became a product) the requirement for these choices to be exposed to developers was intensified. With hindsight, as designers of the system we tended to cater for the optimum configuration for ourselves and this was often inappropriate for others.

- At the time of writing, JTSArjuna is used within 5 separate products and has been sold to 3 other companies to embed within their own products. It is impossible to say with certainty how many users it has, but it has brought millions of dollars to the various companies that have sold it.

- Once we were acquired by Bluestone, the use of JTSArjuna increased significantly and hence so did the support and training load put on the

developers. We quickly realised that a commercial product is much more than the software that actually executes: there is a significant amount of collateral material required too, e.g., training material, white papers etc.

- The biggest mistake we made was in the development of crash recovery for OTSArjuna and tying it to the CORBA model. With hindsight it is possible to see that Arjuna could be used in other, non-CORBA environments and we should have designed accordingly.

- Commercial requirements on robustness of software systems are far more rigorous than you would expect from research prototypes and this was the probably the hardest aspect for us to tackle.

- Working with various standards bodies (OMG, Java Community Process, OASIS to name but a few) has been fruitful but also extremely frustrating at times (committees rarely agree on anything, especially if there is existing product to protect).

The discussion in section three indicates that the original structure of the Arjuna system was sound for its adaptation in various types of middleware. The 15 year journey from academic project to commercial product has been an interesting one and enlightening in many respects. And finally, the most surprising thing has been the amount of use to which Arjuna has been put over the years.

Acknowledgements

The *Arjuna* system is the product of a diverse and varying team over many years. We would particularly like to mention the efforts of Steve Caughey who managed the transition process from academic project to industrial product, Dave Ingham for his work on the JMS, Stuart Wheater for the QA and testing work and Jim Webber for his work on the BTP implementation. The research work at the University was funded by many grants from the UK Engineering and Physical Sciences Research Council, the European Commission and industry that included Marconi, Nortel, IBM and HP.

We would also like to thank the anonymous reviewers and Doug Terry of Microsoft, our shepherd for this paper, all of whose comments helped to improve it.

References

[1] S.K. Shrivastava, G. N. Dixon, and G. D. Parrington, "An Overview of Arjuna: A Programming System for Reliable Distributed Computing," IEEE Software, Vol. 8, No. 1, pp. 63-73, January 1991.

[2] S.K. Shrivastava, "Lessons learned from building and using the Arjuna distributed programming system", Theory and Practice in Distributed Systems, K P Birman, F Mattern, A Schiper (Eds), LNCS 938, Springer-Verlag, July 1995, pp. 17-32

[3] S.K. Shrivastava and D. McCue, "Structuring Fault-Tolerant Object Systems for Modularity in a Distributed Environment", IEEE Trans. on Parallel and Distributed Systems, Vol. 5, No. 4, April 1994, pp. 421-432.

[4] G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M. Little, "The design and implementation of Arjuna", USENIX Computing Systems Journal, vol. 8 (3), pp. 255-308, Summer 1995.

[5] F. Panzieri, and S.K. Shrivastava, "Rajdoot: a remote procedure call mechanism supporting orphan detection and killing", IEEE Trans. on Software Eng. 14, 1, pp. 30-37, January 1988.

[6] G.D. Parrington, "Reliable Distributed Programming in C++: The Arjuna Approach," Second Usenix C++ Conference, pp. 37-50, San Fransisco, April 1990.

[7] X/Open Reference Model, Version 3, X/Open Ltd. 1996.

[8] M. C. Little, S. M. Wheater, D. B. Ingham, C. R. Snow, H. Whitfield and S. K. Shrivastava, "The University Student Registration System: a Case Study in Building a High-Availability Distributed Application Using General-Purpose Components", Chapter 19, Advances in Distributed Systems, Springer-Verlag, LNCS No. 1752.

[9] OMG, CORBAservices: Common Object Services Specification, Updated July 1997, OMG document formal/97-07-04. www.omg.org

[10] Java 2 Enterprise Edition (J2EE) specification, www.javasoft.com

[11] R. Monson-Haefel, "Enterprise Java Beans", O'Reilly & Associates, CA, 2001.

[12] K. Mani Chandy and Adam Rifkin Systematic Composition of Objects in Distributed Internet

Applications: Processes and Sessions. Computer Journal in October 1997

[13] G. Banavar, T. Chandra, R. Strom and D. Sturman, "A case for message oriented middleware", Proceedings of the Symposium on Distributed Systems, DISC99, Bratislava, September 1999, LNCS. Vol. 1693.

[14] Business Transaction Protocol specification, http://www.oasis-open.org/

Tree Houses and Real Houses: Research and Commercial Software

Susan LoVerso, Margo Seltzer
Sleepycat Software
{sue,margo}@sleepycat.com

Abstract

Sleepycat Software develops and supports the Open Source software product Berkeley DB, the most widely deployed embedded database software in the world. Berkeley DB originated at the University of California, Berkeley, and in this paper, we discuss the differences between research software and a quality commercial product. Over the past years we have acquired an education in configuration, portability, and testing. The key message is that code quality, a willingness to rewrite or discard code when necessary, rigorous adherence to internal standards, and constant policing of ourselves are the key requirements of quality software.

1. Introduction

Many research and commercial software communities struggle with technology transfer, converting a research prototype into a commercial product. While the barriers are occasionally political and cultural they are often technical. Regardless, it is a problem research groups must solve to maximize the impact of their research ideas or to profit from their work.

Building software is analogous to building a house: houses should be designed before construction starts, there are building codes to follow, designs and finished products should be reviewed and formally inspected, and the finished product should be flexible enough to accommodate different uses to which customers might put it.

In this analogy, research prototypes are tree houses. While looking like real houses with many of the fixtures of real houses, they are built for a specific purpose: they do not need to last for decades and they do not need to accommodate a variety of uses. There exists a middle-ground between tree houses and real houses that we will call sheds. Sheds are simple, single-purpose structures that must adhere to local building codes (for example, wiring and location). In the rest of this paper, we will use the analogy between house construction and software creation to explore the process necessary to build quality commercial software.

Sleepycat Software develops and supports Berkeley DB, an Open Source database library. Berkeley DB has a long history, starting in the early 1990s at the University of California, Berkeley, where it began as an Open Source replacement [5] for dbm [1] and ndbm [2]. Olson and Bostic added a Btree implementation and a common index-independent API, and the resulting package, Berkeley DB 1.85, was released as part of the 4.4BSD distribution. Using our analogy, Berkeley DB 1.85 was a rather good shed that became widely adopted. It was relatively well-engineered, had consistent interfaces and a consistent coding style, but was lightly tested and supported only a single class of applications needing non-concurrent, unrecoverable data storage.

Seltzer and Olson developed a prototype of a transactional system based on Berkeley DB 1.85 in 1992 [6]. Using our house analogy, it was a tree house; although it demonstrated that a transactional system could be built on Berkeley DB, the system lacked coherent design, shortcuts were taken to avoid solving hard problems, and it was never stable enough to support mission-critical applications.

In 1996, Sleepycat Software was founded to transform the DB 1.85 shed into a real house. While DB 1.85 was in widespread use, it had many shortcomings. The code reflected the multiple authors that had worked on it and there was little consistency between the different access methods. The package worked correctly on common cases, but boundary conditions could fail. For example, the hash package leaked pages in the database if all the items in a bucket were deleted and that bucket had exceeded a single page. There were no utilities to dump and load data from/to the database. There was no way to permit concurrent updates or even a single updater with multiple readers. Even in the read-only case, each process maintained its own cache of recently-used pages, so the system consumed more memory than was necessary. It was a typical shed: it was just great for storing things, so long as nothing catastrophic happened if the things were lost or even slightly damaged.

Today, Sleepycat Software has ten engineers supporting approximately 200,000 lines of code. Berkeley DB 4.X supports Btree, Queue, Hash and Record-oriented access methods, transactions, high levels of concurrency, recoverability, master-slave replication, distributed transaction management, and a wide variety of APIs (C, C++, Java, Tcl, Perl, Ruby, RPC and many more) on a wide variety of platforms (UNIX, Linux, Windows, VxWorks, QNX and more). As Berkeley DB is a library API, it is used exclusively by software developers, not end-users. Our typical sales opportunity is a software engineer who is building an application with data management needs. As we are an Open Source product, we cannot hide our flaws and the engineers who buy our code know exactly what they are getting.

Sleepycat is a small company with limited resources. Generally one or two engineers have responsibility for a project. Those engineers are responsible for all aspects of the project: initial design, implementation, testing, integration and at least a solid first draft of documentation.

Sleepycat is a distributed company. With no fixed office space and engineers on both coasts of the US and in Australia, we cannot walk down the hall and have a spontaneous brainstorming session or a quick whiteboard talk. Good communication is essential to our success. Similarly, various, rigorous processes are imperative. Due to our distributed nature, we have more process than companies orders of magnitude larger. However process is there for maintaining order, not to get in the way of doing the job at hand. Our process must be different from other team-based reviews, such as the Team Software Process (TSP) [8] because we cannot have review meetings other than those conducted over email or the telephone.

Because mentoring is difficult in a distributed company, Sleepycat hires only experienced engineers. Our engineers have all solved difficult problems. We all know how to read and understand someone else's code, and how to make our own code easily understood by others. We understand that code consistency helps us all. We understand that any untested line of code has a bug in it, by definition, and we use any tools we can find to increase our testing and test code coverage.

This paper describes the practices that have been successful for us. Not all of the practices we follow will scale to a large company. The processes we discuss may work well for small teams. Fundamentally, the one requirement necessary is that management must believe that such processes are important and must make them mandatory and part of the culture. For a large company to succeed with these practices it may be necessary to institute a more formal collection of processes and training such as those of the Software Engineering Institute (SEI) [7]. Even with training, large organizations need to incorporate these processes into the everyday workplace constantly. As an organization grows larger, overcoming an individual engineer's resistance to change becomes a bigger issue.

In the rest of this paper we take you through our process of building software and compare it to the construction process of surveying the land (design), offering building choices (portability and configuration), building to code (coding standards), inspections (testing), warranties (customer support), expansion (rewriting), hitting bedrock (unexpected problems), failing inspections (difficult bugs), and preparing to build the next house (lessons learned).

2. Surveying the Landscape

No one would ever let a builder begin construction of a home based on his promise to, "Build you a great 4-bedroom Colonial," yet it is common for engineers to begin coding without a detailed and explicit design. Any software engineering book will tell you this is a mistake, and it would be right. However, such books will often also tell you to use a formal specification language and a great deal of rigor to actually design something. We see things slightly differently.

In the construction of a house, the design is comprised of blueprints, plot plans, artistic renderings of the interior and exterior, an electricity plan, a plumbing plan, etc. The goal is that one person could complete the design and an entirely different team could (and usually does) implement it. While the same should hold true in the software world, it rarely does. Designs in our world consist of a few basic components:

- A goal statement.
  - What is the purpose of the design?
  - What are we trying to achieve?
  - Are there different categories of functionality (for example, features the design must provide and features that are nice but not essential)?
- A technical description of the software to be built.This usually contains:
  - a rough code breakdown,
  - a detailed discussion of the effects on the rest of the system and overall integration with the rest of the product,

    - a description of the algorithms (in our case, special attention is paid to concurrency and recoverability).
- A test plan.
    - How will we test this?
    - Can we use or augment existing tests?
    - Do we need a new testing infrastructure?
    - Do we need to build testing hooks into the code?
- Documentation.
    - UNIX-style manual pages.
    - Reference Guide sections. In our product, the programmatic interface is what we sell, and so documentation is absolutely crucial.

Once the design is finished, the fun begins. Designs are distributed to both engineering and marketing, and while one or two engineers will have responsibility for review, comments from everyone are encouraged. Obviously, sending designs to the entire company does not scale, but having the design read by a group of people with different perspectives is key. Typically, we ask the following questions as we review each design:

- Does the design solve the customer's problem?
- Does the interface share the same look and feel as the rest of the system? An interface that is difficult to learn or explain, or inappropriate for a particular language will not be popular with our customers. (For example, the same services are often provided in different ways in different languages such as C, Java and Ruby).
- Does the interface match interfaces in similar products, and will it make sense to engineers experienced in this area?
- Is the test plan sufficient?
- Does the design scale to next year's hardware, thousands of threads and terabytes of data, or are there performance bottlenecks?
- Does the design affect data recoverability?
- Does the design affect existing customer's software and their existing databases? (Upgrading software annoys customers, but they are usually willing to do so. Upgrading databases is a serious problem for many of our customers.)
- How pervasive is this change? Can we introduce this change at this time in our release cycle and still reach stability in time for the next release?

The design phase lasts until everyone's concerns and perspectives have been addressed. Coding will often begin before the design has been finalized, but with the understanding that sunk costs are irrelevant, and it is better to be right than to be done quickly. Consensus is almost always reached, but answers are occasionally dictated by the product architect (and official arbiter of good taste).

When the design phase is complete, we have a workable design and schedule. Nonetheless, the development engineer always needs to change things as the coding proceeds. Just like code is audited before it is committed, design changes are propagated to the review engineers for comment. In general, we have not had big surprises when we follow our design protocol rigorously. However, it has certainly been the case that when we are sloppy during the design phase, we suffer for it later.

3. Offering Building Choices

Just as a treehouse is usually built to fit the contours of the particular tree that is in the yard, research prototypes work only on the specific systems and configurations for which they are built. Sheds are similarly built with the confines of the particular yard in mind. They are small enough to be nimble in their placement. A developer cannot hire an architect to design and build a brand new and different house on every lot on which the builder builds. Therefore, a developer has a selection of house designs, each of which can be altered in limited ways to suit the designs to a homeowner or site, such as reversing or rotating rooms around the core or building the garage at basement level. Commercial software must have the flexibility to be configurable and portable to many requirements and systems.

To the extent possible, projects should choose a development language that minimizes the differences of the underlying environments in which it will run. C or C++ projects can be made portable more easily than Fortran. Java is preferable to C/C++, and a high-level scripting language such as Python or Ruby is the best choice of all. Berkeley DB was forced to use C: it was difficult and time-consuming to make C++ run as quickly as native C and when Berkeley DB was developed there was no C++ standard and available C++ compilers varied in major ways. Also, until quite recently, it was impossible to make Java run as quickly as native C.

Portability code should be isolated to a single area and a single set of files. For example, in Berkeley DB, there are `os` source directories (currently `os`, `os_win32` and `os_vxworks`). Compiler, library and operating system interfaces with portability issues are abstracted to files in the `os` directory. This includes variables such as

errno, library interfaces such as malloc, and operating system interfaces such as mmap. The portability layer of Berkeley DB is approximately 3000 lines of C.

There are two advantages in creating a portability layer. First, it is easier to port to new platforms. It is often the case that no member of the development team knows the porting platform (for example, there are literally hundreds of different embedded operating systems, and nobody knows any significant fraction of them). By creating a separate portability layer it is possible for someone intimately familiar with the port platform to port Berkeley DB without having to understand Berkeley DB itself.

Second, software rarely needs the full functionality of more complex system calls such as mmap, and programmers commonly configure such complex interfaces incorrectly. A portability layer allows us to export only the limited, necessary functionality from the system, simplifying the code in Berkeley DB. For example, code to use the UNIX stat system call to determine if a file exists and its optimum block size for I/O is complex, as the stat field which holds I/O block size information is a relatively recent addition to UNIX. Also, the code to use stat to determine if a file exists and is a directory is complex, involving the manipulation of octal byte masks on many historic systems. In the Berkeley DB portability layer we have written that code, but exported it to the Berkeley DB mainstream code as two functions: __os_exists and __os_info, which only return the specific information needed.

For C and C++, language types can also be a problem (for example, using size_t and off_t portably can be difficult). In Berkeley DB, we define internal types of fixed size in our software and abstract them through the portability layer APIs. For example, rather than deal with the fact that an off_t may be 16, 32 or 64-bits on a particular platform, we use "page number" and "page size" variables in our software, both of which are declared to be of type uint32_t to guarantee us 32 unsigned bits, and only convert to the platform-dependent off_t when making the system call in the portability layer.

Portability code should always be selected based on a feature, and never based on a platform. Trying to create a separate portability layer for each supported platform results in a multiple update maintenance nightmare. A "platform" is always selected on at least two axes: the compiler and the library/operating system release. In some cases there are three axes, as when Linux vendors select a C library independently of the operating system release. With M vendors, N compilers and O operating system releases, the number of "platforms" quickly scales out of reach of any but the largest development teams. By using standards such as the ISO/IEC C-language standard[11] (ANSI C) and the ISO/IEC system call API standard[12] (POSIX), the set of features is relatively constrained.

Of course, the default portability code should implement whatever "standard" is most widely available on supported platforms. For C-language applications this will likely be the previously mentioned ANSI C and POSIX standards, at least on platforms shipped in the past decade. The kinds of problems we solve in the portability layer are when systems fail to match the standardized behavior, or differ from the standard for some other reason. For example, the dosFS filesystem distributed with VxWorks does not support the POSIX-mandated semantic that software be able to open a file descriptor, remove the underlying file, and then continue to write to the open file descriptor. We solved this problem by adding a flag to the Berkeley DB structure that contains file handles (currently, either a POSIX file descriptor or a Win32 HANDLE). The flag allows the portability layer code that closes file handles to remove the file after the last close of the file handle. As a result, only the portability layer code needs to handle this problem, and Berkeley DB programmers do not even know that it exists.

Portability choices can be made along either lines of code or compiled files. We have not found any maintainability differences between using compile-time tests to include alternate lines of code, or compile-time tests to use one of a few different files. Generally, we have moved portability code for different platforms into separate files when the implementations diverged significantly (shared memory mapping on UNIX vs. Windows) and left it in a single file when the differences were minimal (using gettimeofday, clock_gettime, ftime or time to find out the current time-of-day). When a platform has separate files, we create a new directory but leave the file name as close to the generic file name as possible. For example, absolute path name resolution for all platforms except VxWorks and Win32 is in os/os_abs.c, while the VxWorks implementation is in os_vxworks/os_vx_abs.c and the Win32 implementation is in os_win32/os_abs.c. The VxWorks filename is different due to a workaround for a Wind River build problem.

Configuration choices should be made at compile-time. A significant advantage in distributing source code is that it allows the package to adapt at compile-time to the

environment it finds. This is critical as it allows the package to run on platforms its developers have never seen, and it allows the software's community of users to do their own ports. Only the largest of development teams can afford to buy all of the hardware and hire enough employees to support even a limited number of platforms.

Berkeley DB uses the GNU Autoconf software to do compile-time configuration. Autoconf is a tool for producing shell scripts that automatically configure software source code packages to adapt to many kinds of UNIX-like systems. The configuration scripts produced by Autoconf are independent of Autoconf when they are run, so their users do not need to have Autoconf. While Autoconf scripts are difficult to write and difficult to maintain, Berkeley DB does all its configuration testing for UNIX and UNIX-like systems in around 1300 lines of Autoconf script.

Regardless of your approach to configuration, do not ask the user for system information. The user installing the package does not know the answers to your questions, and this approach is doomed from the start. The software must be able to determine for itself any information that it may need at compile- or install-time.

Finally, portability and configuration issues are greatly diminished by following good software practices in other parts of your development:

- Having a good test suite allows the team to buy inexpensive hardware for testing and then easily run regression tests before releases.
- Distributing the test suite allows the community of users to test their ports before contributing them back to the group.
- Never using the namespace of any other part of the system, (including file names, error return values and function names) ensures that you never collide with another library. Where the namespace is shared, documenting the namespaces you use is only courtesy.
- Encouraging developers to use a wide variety of platforms as their desktop and test machines ensures that the code is always being tested for portability flaws.
- Aggressively keeping the code base clean and the software layers independent makes it less likely that portability problems will slip through.

4. Local Building Codes

As a builder must comply with local zoning and building codes, so must software developers observe company coding standards. When building a tree house, there are rarely zoning laws or permits, and the tree house takes whatever form the builder chooses. Similarly, in a research prototype, the code is in whatever language and form the developer prefers. With a shed, there are some guidelines or local laws. Similarly, when engineers cooperate on a project, they generally work more closely together and use the coding standards, if any, of the larger organization of which they are a part, just as a shed might be painted to match a house. If those standards are inconvenient for some reason, however, they will often be ignored.

When building a real house, there are many participants. The excavator must dig the foundation to match the footprint of the house, the carpenters must build the walls to match the blueprints, etc. There is typically a general contractor overseeing the operation to make sure the subcontractors do their particular jobs correctly and match the specification. Similar processes and policies are necessary for building quality commercial software as well.

The debate over the choice of coding standard could probably go on forever. Regardless, it is simply too difficult and expensive to maintain software built using multiple coding standards, and so some coding standard must be chosen. However, a coding standard is useless without enforcement. Consistency and cleanliness of the code are of the highest priority; the details of the standard itself are a distant second. Not only does Sleepycat have a single standard for coding, we adhere to it religiously. Code audits frequently contain suggestions about clarity, proper spelling and punctuation in comments, points of line-splitting, choice of variable names, and a host of other things that have little to do with the technical correctness of the code, but everything to do with the quality of the overall code base. Newly hired engineers are usually taken aback by the rigor with which we strive for consistency, but over time, they realize that adherence to the local codes is not an option; it's the law.

Code consistency makes it easier to maintain cleanliness in the code. Bug reports and user questions are assigned to engineers not only on a knowledge-area basis, but also on a time-available basis. As a result engineers must quickly be able to understand code with which they are not familiar. Fundamentally, reading another engineer's code is reverse engineering. Consistent use of whitespace and punctuation makes it possible to concentrate on what the code does, rather than how it is formatted. Consistent variable naming (for example, across our code base, a database handle is always a `dbp`)

makes it possible to immediately understand the set of operations possible using a particular variable. Not only is consistency and code cleanliness necessary for our daily work, it is necessary for our success as an Open Source company. If a builder built homes that did not meet code or zoning laws, or if the walls were not vertical and straight, that builder's reputation would be tarnished, and business would go elsewhere. The quality of the code matters a lot in an Open Source product. Good code gains you credibility and respect. Putting thought into organization and consistent structure leads people to believe that the same care goes into the code itself, which we believe to be true in our case.

Conversely, sloppy code, or code that produces warning messages when it is compiled, or spelling errors in README files or error messages, leads customers to believe the engineering behind it is also haphazard and sloppy. We regularly have customers comment on the quality of our code: "I must say, your code is excellent," or, "It is a real pleasure to read." That feedback not only strokes engineering egos, it is every bit as important to a software development organization as it is for a builder to hear, "Your workmanship is excellent."

Maintaining consistency at Sleepycat is easier than it is at most organizations due to our size and our having an experienced and disciplined engineering team. However, we have processes in place to catch mistakes: all code changes are audited, and audits include checks for stylistic violations. Before a release cycle, we make a pass over the code looking specifically for coding standard and naming mistakes and spelling errors in quoted strings. Compilers are run with "additional warning" flags and any warnings found are cleaned up. We regularly run code-cleaning tools such as lint and others. All engineers are expected to fix violations they find when reading code, regardless of why they were reviewing the code in the first place.

5. The Home Inspection

Typically, completion of a tree house consists of the builders declaring, "It's done!" and 30 seconds later, the children are using it. When a shed is completed, there might be a quick inspection by the local housing authority, and the homeowners can then move their lawn mower inside. When a house is completed the process is more involved. With a reputable builder, the process is verified on a recurring basis so there are not any surprises when the building inspector checks the finished product. After receiving certificates of occupancy from the local authorities, the builder does a formal walk-through of the residence with the new homeowner. New owners perform unit testing on the house: turning on lights and water, flushing the plumbing, and so on. It is the same way with commercial software. Research prototypes rarely undergo rigorous testing, and need only work in a limited setting. Larger software projects might undergo some testing, but are still often expected to be extensively fixed after release. A commercial software product may not survive if it is filled with severe flaws upon release. Maintaining buggy, released software costs the industry billions of dollars every year [4], so every bug found in testing translates into long-term savings.

Given our small support organization, the high level of algorithmic complexity in our code, and the demands of our customer base to never lose data, we live and die by testing—we simply cannot afford to debug problems in the field and must find them before release. We currently have over 40,000 lines of Tcl, testing Berkeley DB's functionality. Of course, we run code coverage tools over test suite runs so we know what areas of the code are not getting covered by our tests. Currently our Tcl test suite covers about 70 percent of our code base by lines of code (a good level, given the difficulty of exercising error code paths in the typical application). Our test suite is composed of hierarchical layers:

- At the lowest layer we have tests of a particular function in the system (for example, does the system handle items that exceed the page size).
- At the next layer, we wrap the lowest layer tests in drivers that run each access method through every test with a variety of parameters (for example, page size, encryption on or off, and so on).
- At the next layer, we test different system configuration options: transactions, locking, replication, RPC. For each configuration, we run all tests in the lower layers.
- At the next layer, we invoke the different tests through each of the high-level APIs (C, C++ and Java).

Not surprisingly, the test suite takes days to run. It is modular enough that engineers can run pieces of it to test code changes, but producing a release implies running the full test suite on a wide variety of compiler and operating system combinations. However, every bug we find in testing is a bug customers do not find, and in a system as complex as ours, many of the bugs we find simply could never be debugged in the field.

However, our test suite, written in Tcl, has limitations. Standard Tcl is not multi-threaded and therefore our test suite is serial as well. In addition, scripting languages are still too slow to stress the library's performance. For these reasons, we have two large application server pro-

grams (intended to mimic customer application behavior) that we run continuously on high-end test hardware. The layers of unit testing described above test the library's functionality. The test applications test the library's performance and concurrency control, as well as resistance to non-deterministic behavior, such as random system failure. In an earlier release, our most serious mistake was failing to run our test applications during the development cycle, resulting in an unexpected slip in the release. Tens of thousands of successful iterations of our test applications is what gives us the confidence that a new release is ready for real-world workloads.

6. Warranties and the New Homeowner

When a general contractor builds a house, and the new owners move in, there is a period of time during which the builder will come back and fix anything that is not working or is not quite right. In our business, this is called customer support. Sleepycat has a support organization to track requests and direct them to the appropriate people. However, once the request has been handed off to an engineer, the customer and engineer are in direct contact. This is for several reasons:

- Potential customers are likely evaluating our competitors' products while waiting for us to respond. We need to respond quickly and correctly and not let support requests languish for too long.
- Engineers often select products based on the quality of the support. If they believe the vendor organization will support them as they learn a new API, they have a strong incentive to buy from that vendor.
- We are selling engineering software to smart people. It would be difficult to train support people that could handle a significant fraction of our support questions without being engineers.
- It is important that engineers understand who pays their salary. Contact with customers is an important source of feedback both on future features for the product as well as how the software is perceived, that is, how well the engineers are doing their job.

Sleepycat uses all of its engineers (including our chief architect) in support tasks. For this reason the quality of our support eclipses our competitors, and no single engineer is tasked with the boring work of support. Engineers cannot close support requests; the close must be done by the support manager, who has responsibility for tracking requests and ensuring that the customer is happy at the end of the day. Our engineers behave like a professional support organization with the directives:

- Be polite (no matter what).
- Respond as quickly as possible; support is more important than new code.
- Answer the question as completely as possible.
- Never send out an untested patch to a supported customer.
- Include a link to the appropriate web page if you give them a documentation reference.
- Be grammatical, spell-check all email, and format it so it is easy to read and easy to reply. Customers assume that sloppy email reflects sloppy code.

Sleepycat also treats requests from unsupported users similarly to requests from supported customers, although at a somewhat lower priority. Often, unsupported users find bugs and have good ideas for features. We also believe that helping academic and other unsupported users become comfortable with our product and API will result in future commercial sales.

Sleepycat maintains a complete log of all interaction with customers on every support request. We have had cases of customers wondering what had happened with their support request and our log has been useful in helping them identify errant email forwarding and other black holes into which email fell. In general, our guiding principle is that customers desperately want to talk to a technically competent human if they are reporting and tracking a real bug, and it is our job to provide that human being and a resolution as quickly as is possible.

7. Starting to Dig the Expansion

After you have lived in a house for a while, sometimes your needs change. The family size grows, new room uses are sought. Initially, for a growing family, one may take the faster solution of, say, finishing part of the basement and turning it into a playroom. However, depending on the growth, ages and needs of a family, more drastic measures may be needed. Aging grandparents may be moving in and their needs are different from a toddler's. It might be time to reconfigure some of the existing rooms and add an addition for the changing needs. So it is with software and rewriting particular subsystems.

There has been much discussion in the community about the pitfall of rewriting entire systems from scratch [9], however there has been little discussion concerning the trade-off between rewriting and maintaining subsystems that were written under different conditions than currently exist. As one maintains a product over time, one discovers parts of the code that have become

complex, brittle, and/or simply ugly. The code did not start out that way, but as features were added, fundamental assumptions upon which the code is based ceased to be valid. For example, when we built the portion of our system that handled the recovery of file creation, the system observed the rule that every file contained one and only one database. Sometime later, we added subdatabase functionality that let us store multiple databases in a single file. Suddenly all the assumptions we made about the state of a file during an open were no longer valid. Then we added replication to our system which further changed the rules about opening and creating files. With much painful debugging, we were able to make the code pass our tests, but the result was a largely unmaintainable codebase. We had a storm brewing in the code; nobody admitted to really understanding the code, and engineers were loathe to touch it for fear of breaking what seemed to work.

It was time to rethink our entire approach and rewrite this part of the system based upon the new state of the world. Since we were doing that, we had the luxury of considering new functionality as well, and we decided that we could extend our transactional support to provide the ability to group multiple file system level operations in a single transaction (e.g., begin, create file foo; delete file bar). It is terrifying to contemplate rewriting parts of the system that work, but even more terrifying to contemplate leaving fragile code in your product. The key enabler allowing us to rewrite this central component of our software was our test suite and testing infrastructure. Without the test suite, we could never have considered this rewrite. We began the rewrite with a new set of assumptions and designed it based upon a simple set of four file system primitives dealing with modification:

- create files
- remove files
- rename files
- write to files

The design phase lasted about a month with a single engineer owning the design and the rest of the team commenting on it and pointing out the problematic cases. Each engineer brought their own set of biases, assumptions, and knowledge about particular subsystems to the table and the end result was that we debugged significant parts of the design before the first piece of code was written.

The rewrite took about one staff-month of actual coding, one staff-month of unit testing, and a final staff-month of new testing, that is finding and fixing bugs that were discovered as the test suite was enhanced to cover the new functionality that was added. The end result of this rewrite was increased functionality and cleaner code that we no longer fear touching. It was neither easy, painless, nor fun, but it was important and the end result is enormously satisfying.

Perhaps the most educational aspect is the constant tension between doing it right from start to finish and having to rewrite parts of the new code as you go. As the development progressed, there were phases where certain functionality was working, but we would discover a problem that required changing the working functionality. The engineer on the project always resisted such changes, and it took strong and persistent management pressure to ensure the end result was not as brittle and complex as the original code. In general, engineers resist changing things they have written and debugged with good reason; however, sunk costs are irrelevant if the path you are on is headed for disaster. It is absolutely essential to be willing to backtrack. The engineer involved may not realize when it is time to backtrack, which is why someone else must be close enough to identify the need.

8. Hit Bedrock, Stop Digging

Suppose you decide to add an in-ground pool in your backyard. Someone from the pool company comes to your house and surveys the yard, determining where various obstacles might be. Everyone agrees on the pool design and area and the price to install it. Work merrily commences. However, after digging down several feet, the excavator hits bedrock. Work ceases and the options must be considered. Should the bedrock be drilled and blasted (with the unlimited expense paid for by the homeowner) or should the pool be moved, or should the pool just be shallower than originally planned?

Sometimes design flaws are only discovered during implementation. It is important to be willing to admit, at any time during the life of a project, that the wrong path has been chosen and that it is time to step back and rethink the entire design. The brute force alternative, of hitting the round peg really, really hard until it goes in the square hole, will be more expensive in the long term.

Sleepycat recently had one such experience when adding checksum and encryption support to our database product. All I/O in the product is done at a page-level granularity, so we chose to checksum and decrypt/encrypt each page as it was read/written. Cryptography support requires we store 36 bytes of crypto initialization vectors (IV) and checksum data on each page of the database.

The design had to satisfy two additional criteria:

- We did not want to use 36 bytes of space per page unless the user wanted checksums and encryption, as 36 unused bytes on a 512-byte page is a significant overhead.
- We did not want to change the physical page layout written to disk because customers using this release would then have to upgrade their databases (even if they did not need the checksum and encryption functionality).

A Berkeley DB database page begins with a collection of header information (25 bytes), and then has an array of page indices referencing data items on the page, growing forward. The actual page data begins at the end of the page and grows backward. Much of the Berkeley DB software manipulates page structures: getting a reference to bytes at a specific index, computing the first free byte on the page, computing the last free byte on the page, computing the remaining free space, and so on.

In the original design, the engineer decided to retain the original page header. The idea was to put the checksum and crypto information at the end of the page. That would mean only the free space calculation code would need to be adjusted, and code dealing with indices and other parts of the page would remain unchanged. The design did not modify the page header at all, so no upgrade would be necessary. This design was simple, clean, and passed review. Implementation was conceptually simple, as the bulk of the changes were in one header file and one page-related file. After implementation, initial testing of both checksum/crypto pages and standard pages passed. The code was committed into the main source tree.

However, when full test suite runs were made, things started breaking. The problem was that the page size was used for many calculations that were not immediately obvious. Also, there was an assumption in the code that the end of the page is the end of the page's data space. The immediate reaction was to begin bug-fixing and bandaging the code, and adjusting the computations using the page size. However, after some days of this effort, the engineering group remembered the first rule of holes: "If you find yourself in one, stop digging." It was time to climb out of the hole, revisit the design, and change direction.

The alternative solution was to place the checksum and IV at the end of the page header but before the array of indices. The coding changes were larger than the original solution as the location of the database page indices was no longer fixed. However, the necessary code changes were largely flagged by the compiler, and the solution no longer broke in subtle ways: it either worked or broke dramatically.

Coding the original paging scheme, testing it, attempting to fix it, discarding that change, and then repeating the procedure with a new solution exceeded the original schedule estimates. However, in the long term, the code is far more maintainable, and that must be the primary goal. The lesson is that even careful review is not always sufficient, and engineers must know when to withdraw, regroup and plot a new strategy, incorporating the knowledge gained.

9. Failing Inspection

Just because a building inspector looks at a house and grants a certificate of occupancy does not mean that all of the work was performed perfectly. You may find that although a gas line is installed correctly and up to code, its placement is off when you actually attempt to hook up the clothes dryer. Or that although the walls look straight, they are noticeably crooked when furniture is installed. So it is with software: even with testing, reviews, checks and balances, bugs are found because users use the software in unanticipated ways.

One customer reported a thread starvation problem when several equal priority tasks performed database operations and a lower priority task performed checkpoints. The problem was that when the checkpoint task wrote a dirty buffer, it set a flag in the buffer indicating that I/O was in progress. However, the checkpoint task was pre-empted by a higher priority database task. If the database task needed to access the page in the buffer being written by the checkpoint task, starvation occurred. However, Sleepycat found and fixed exactly this starvation issue five years ago, in 1997. In our fix, the database task, seeing that I/O was in progress, released all of its locks and relinquished the CPU, letting the checkpoint thread continue. So, why was this happening again?

Unfortunately, the original fix only worked on systems where the competing tasks were of equal priority or the system was lightly loaded. This report was on a true real-time system, where all high priority database tasks were given the opportunity to run when the one task yielded the processor, and so the checkpoint task was never able to run. Understanding the fundamental problem was challenging because the fix was already in the code and we needed to expand our thinking to the real-time space to understand why it still failed. The lesson is that software is intimately related to its environment, and reliable software must be both general in nature and

flexible, as the user's environment will always differ from the developer's environment.

Another recent challenging problem occurred while running our test suite on an embedded system. A handful of tests were taking an assertion while acquiring a self-blocking mutex lock, because the locking code was unexpectedly returning an EDEADLK error. This particular code is one of the few places where we use self-blocking mutexes.

In DB, the code to allocate and initialize a mutex takes an argument for flags. Some of the flags affect the mutex, such as the one indicating that this is a self-blocking mutex. Some affect the allocation code such as one indicating whether the shared memory region (where the mutex is allocated) needs locking or is already locked by the calling function. Therefore, the mutex code looked like this:

```
if (we need a new mutex){
   __db_mutex_setup(..., &m,
      (SELF_BLOCK |
      is_locked ? NO_LOCK : 0));
   MUTEX_LOCK(m);
}
MUTEX_LOCK(m);
```

It was the second call to MUTEX_LOCK that returned the EDEADLK instead of blocking as expected. So, why was this failing, and only on this one system and nowhere else? The possibilities included:

1. This system used pthread mutexes. Most systems we have use test-and-set mutexes. Perhaps there was a bug in our Pthread mutex code.

2. Since self-blocking mutexes were not frequently used, perhaps we were hitting a bug in the system's pthread implementation.

3. It was something else.

Given that the test suite was only failing on this system and no other in this way, our tendency was to think option #2 was the most likely cause. Option #1 was a possibility but that code is extremely stable in DB and has been virtually unchanged for years.

Fortunately, we have a multi-threaded mutex test application that directly calls the DB mutex code. After easily porting that to the embedded system, and many successful runs, we concluded that the mutex code worked as expected (both DB's and the system's) and the failure must be due to option #3 above and we were almost back where we started. Additional, fairly painful, debugging yielded the true bug, and it is in the code snippet above. The bug was that the SELF_BLOCK flag was never getting passed into the setup function, due to a misplaced parenthesis and different precedence. The correct code must read:

```
if (we need a new mutex){
   __db_mutex_setup(..., &m,
      SELF_BLOCK |
      (is_locked ? NO_LOCK : 0));
   MUTEX_LOCK(m);
}
MUTEX_LOCK(m);
```

Debugging on this particular embedded platform is not very easy. So working through this problem was more difficult than it would normally be. After working through this problem a few questions needed to be answered.

1. **Why did this problem only show itself on this one system and nowhere else?** Almost all other systems use test-and-set mutexes, which don't use the pthread code. The test-and-set code ignores the SELF_BLOCK flag. The other system we have using pthread mutexes used a different code path.

2. **What did we learn?** The lessons learned here are that it is important to run the test suite on every system possible and follow up vigorously with all problems. A few times during this debugging, which took a couple of days, we were ready to simply assume it was a system problem and move on. Thankfully we resisted that urge.

10. Preparing to Build the Next House

Just as builders should review their experiences after completing developments, software developers must also review their projects. If a builder built in a town with strict codes and the builder successfully adapted to those codes, s/he might consider retaining those practices even in the face of more lenient codes in another town. Doing so will likely garner the builder a reputation for building a high-quality product, and it's always simpler to have one process in place than many. In software, our lesson for coding standards is that high-quality, clean code is a matter of constant, diligent practice. You cannot start sloppy and end clean. You need to incorporate the cleanliness from the beginning.

A builder may find that of the handful of house designs, several are never chosen by customers and new, expanded designs are required. In our software, when the subsystem no longer meets the needs and requirements of current problems, it is time to consider replacing that subsystem, knowing the experience that went

into it in the first place. You can tell it is time to consider such a measure if you are not making reasonable progress on bug fixing or new features. If the subsystem has gotten too complicated, requirements have changed, or the design was wrong, you will usually find that normal maintenance becomes increasingly difficult. That is a good sign that it is time to start over. A common lifecycle for software is the gradual, constant addition of code, until the software is so unmaintainable that it can no longer be supported or developed except through the efforts of armies of low-paid, brute-force testers. Eventually, everyone throws up their hands, retires the product, and starts with a clean sheet of paper. This evolution doesn't have to happen: if the software has been well-maintained and re-architected as needed, with old features being discarded and removed from the code as new ones are added, the code base may never have to be rewritten from scratch [10]. Absolute compatibility will not happen, of course, but necessary compatibility will.

11. How to Become a Builder

Someone who has built a treehouse or a shed might decide they really want to become a builder and build a house. While s/he may know how to make a hammer meet a nail, there are larger issues to deal with in order to be successful in that transition.

We have several suggestions for researchers who want their code widely used. There should be no surprises in this list; it simply summarizes the points that have been made throughout the paper.

1. Write documentation. The source code is not self-documenting and code written by a myriad of independent students or researchers needs to be documented in a traditional sense. Users attempting to use your software need written guidance and specific instructions.

2. Choose and enforce a coding standard. It will teach students an important lesson early on, and it will make life easier for all students and others coming later. Also, people using your code will be able to read it and navigate through it, because the code is consistent and predictable.

3. Invest in release engineering so that a user can easily download your code and run it, anywhere. That can mean building binaries, using Autoconf, or building your own configuration system.

4. Be willing to answer questions and help people get over the initial hurdles of using your software. What is obvious to you because you've thought about it for the last several years may not be obvious to someone else immediately. There is a tendency to become irritated with trivial questions and assume that the "intelligent" users wouldn't ask such things. This is an enormous mistake; while these trivial errors might indicate that the user has not read the manual, more frequently, they indicate that something is not documented or is documented, but confusing. Treat user questions as you would paper reviews and ask yourself, "What information does this person need that s/he was unable to get from our documentation?" Being responsive to user input and questions will require personnel who are responsible for the task.

5. Clean up the code after you've finished writing a paper. The act of publishing tends to leave the code littered with quick hacks that were necessary to run the right tests at a particular time.

The constant tension in all this is that academia places little value on these activities, so it is difficult for academic researchers to devote the time, energy, and finances to this process. The tension might be somewhat less in an industrial setting, but industrial research groups do not always have the personnel or resources to devote to the process either.

12. References

[1]. AT&T, DBM(3X), Unix Programmers Manual, Seventh Edition, Volume 1, January 1979.

[2]. Berkeley Software Distribution, NDBM(3), 4.3BSD Unix Programmers Manual, University of California, Berkeley, 1986.

[3]. Berkeley Software Distribution, DB(3), 4.4BSD Unix Programmers Manual, University of California, Berkeley, 1994.

[4]. RTI Health, Social, and Economics Research, The Economic Impacts of Inadequate Infrastructure for Software Testing, RTI Project Number 70071.011, NIST Planning Report-02-3, May, 2002, `http://www.nist.gov/director/prog-ofc/report02-3.pdf`.

[5]. Seltzer, M., Yigit, O., A New Hashing Package for UNIX, Proceedings of the 1991 Winter USENIX Technical Conference, Dallas, TX, January 1991, 173–184.

[6]. Seltzer, M., Olson, M., LIBTP: Portable, Modular Transactions for UNIX, Proceedings 1992 Winter USENIX Conference, San Francisco, CA, January 1992, 9–26.

[7]. Software Engineering Institute, "Building High Performance Teams using The Team Software Process and Personal Software Process," Carnegie Mellon Uni-

versity, http://www.sei.cmu.edu/tsp, January 20, 2002.

[8]. Software Engineering Institute, "The Team Software Process," Carnegie Mellon University, January 20, 2002, http://www.sei.cmu.edu/tsp/tsp.html.

[9]. Spolsky, J., Things You Should Never Do, Part I, Joel on Software, Apr 6, 2000, http://www.joelonsoftware.com/articles/fog0000000069.html.

[10]. Spolsky, J., Good Software Takes 10 Years, Get Used to It, Joel on Software, July 21, 2001, http://www.joelonsoftware.com/articles/fog0000000017.html.

[11]. ISO/IEC 9899:1999: Programming languages — C

[12]. ISO/IEC 9945-1:1996: Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language].

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *;login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see *http://www.usenix.org/membership/specialdisc.html* for details.

USENIX & SAGE Thank Their Supporting Members

USENIX Supporting Members

❖ Atos Origin B.V. ❖ Freshwater Software ❖
❖ Interhack Corporation ❖ Microsoft Research ❖
❖ Motorola Australia Software Centre ❖ OSDN ❖
❖ Sendmail, Inc. ❖ Sun Microsystems, Inc. ❖
❖ Sybase, Inc. ❖ Taos: The Sys Admin Company ❖
❖ UUNET Technologies, Inc. ❖ Ximian, Inc. ❖

SAGE Supporting Members

❖ Certainty Solutions ❖ Collective Technologies ❖
❖ ESM Services, Inc. ❖ Freshwater Software ❖
❖ Microsoft Research ❖ OSDN ❖ Ripe NCC ❖

For more information about membership, conferences, or publications,
see *http://www.usenix.org/*
or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: *office@usenix.org*

ISBN 1-931971-05-6